Process Algebra Approach to Parallel DBMS Performance Modelling

Chai Seng Pua

Thesis submitted for the degree of
Doctor of Philosophy
Department of Computing and Electrical Engineering
Heriot-Watt University
Edinburgh

April 1999

Declaration

I declare that this thesis was composed by myself, and that the work it presents is my own, except where otherwise stated.

Chai Seng Pua

Table of Contents

Appendixes

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor Prof. M. Howard Williams for guiding me in my research on performance modelling particularly in parallel database systems. I also like to thank him for his support and advice for the past few years.

Thanks are also due to the people in the Database Research Group for their assistance. Also, thanks to ICL for granting the permission to conduct some of my experiments on STEADY. I should like to thank the Laboratory for Foundations of Computer Science, University of Edinburgh, for access to the PEPA workbench.

Never would I forget the support and encouragement from my parents, my fiancée and friends. Last but not least, I should like to thank to Universiti Putra Malaysia for the funding that has enabled me to complete my work during the last three and a half years.

Abstract

With the increasing interest in using multiprocessor computers as database servers, there is a corresponding interest in performance prediction of parallel database systems. Both simulation and analytical approaches have been used and reported in literature. This thesis reports on an investigation into how a stochastic extension to a classical process algebra known as PEPA can be used to model the performance of a parallel database system.

PEPA is a mathematical formalism that provides a small but powerful set of combinators that allow a system to be described in terms of the interaction of its components. The investigation starts with a simple database system running on a simple platform, supporting a simple workload. As the investigation progresses, the system is extended gradually. In doing so, the problem of state-space explosion, which happens when the model becomes bigger and more complicated, has to be overcome. An adapted form of the decompositional evaluation approach and the concept of intermediate threading are introduced to solve this problem.

The investigation builds up to modelling actual parallel database systems. For this purpose, Ingres Cluster DBMS and Informix XPS DBMS, two commercial database systems were studied, incorporating both inter-transaction and intra-transaction parallelism. The results obtained from the investigation are compared to those produced by an analytical approach.

Chapter 1

INTRODUCTION

1.0 Rationale

Parallel database system performance modelling has begun to attract the attention of researchers as a result of the increasing demand for using parallel computers as database servers. Both simulation and analytical approaches have been used and reported in the literature. On the one hand, the simulation approach requires a substantial amount of computing time to obtain a result [69]. While on the other hand, the analytical approach needs complicated analysis in order to produce an outcome [112].

In this thesis, a stochastic extension to the classical process algebras, Performance Evaluation Process Algebra (PEPA) [53], is used to model the performance of a parallel database system. Process algebra is a mathematical formalism that is used to model communication and concurrent systems. It offers the compositional ability that allows the performance modelling to be done in a structured fashion. Although PEPA is not as well established as other formalisms such as Petri Nets [92], stochastic Petri Nets [5] and queueing theory [70], it has been used in modelling the performance of real time systems since then. Nevertheless, we believe that the idea of using PEPA to model the performance of a parallel database system has not been attempted before.

This thesis shows how a mathematical formalism can be used as an alternative tool to model, evaluate and verify the performance of parallel database systems. This chapter presents a brief overview of the thesis.

1.1 Thesis Overview

Chapter 2 presents a general background to parallel database systems. As the mainframe designers begin to encounter difficulties in coping with the increasing demand for powerful machines, others have begun to find a replacement for mainframe computers in the form of multiprocessor computer [35]. There are three common multiprocessor computer architectures. Each of these architectures has its strengths and weaknesses. As the number of processing elements in a machine increases, it is important to maintain a good load balance across the system in order to produce good performance [29]. This leads to a discussion on data partitioning and transaction parallelism. System consistency and reliability are equally as important in parallel database system as are high performance and high availability. For this reason, a concurrency control mechanism is reviewed. Many database machines have been developed and a few of these are discussed in this chapter.

Chapter 3 discusses the general concept of performance evaluation and prediction. Several techniques that are used to measure and estimate system performance are reviewed. In addition to this, several benchmarks that are used to test the performance of computer systems are also reviewed. This chapter continues the discussion with some works related to the system performance evaluation and prediction particularly for parallel database systems.

Chapter 4 presents the background to PEPA (Performance Evaluation Process Algebra) [53], which is the tool used to study the performance of parallel database systems in this thesis. In PEPA, a system is expressed in terms of the interaction of its components and the activities in which these components are engaged. PEPA provides a small but powerful set of combinators which allow the expression of the activities to be constructed in a structured fashion. The syntax and semantics of the PEPA notation is presented. However, it is not sufficient simply to express a system in PEPA notation. The notation needs to be transformed into a matrix that is subsequently used to estimate the desired result. For this reason, the system performance evaluation procedure is explained. The chapter concludes with related work.

After this introduction to parallel DBMS and PEPA, Chapter 5 considers the general approach to using PEPA to model the performance of a parallel DBMS. This chapter starts the investigation with a simple database system running on a simple platform, supporting a simple query. The system is mapped into a PEPA model and the performance of the model is evaluated. The main drawback of PEPA is the state space explosion that occurs as the model becomes more complicated or bigger. Thus, model simplification or model decomposition is necessary to reduce the state space of the model sufficiently to converge on a solution. Chapter 6 continues the discussion with multiple nodes with a single disk system. A technique known as intermediate threading is also presented.

In Chapter 7, PEPA is used to model Ingres Cluster DBMS and Informix XPS DBMS. The first system utilises inter-transaction parallelism whereas the second utilises both inter-transaction and intra-transaction parallelism. The study focuses mainly on the utilisation of the transaction processing parallelism and consequently both commercial parallel DBMSs have been modified to suit the purpose of the study. A series of experiments are reported. The results of the experiments are compared to those obtained using an analytical throughput estimator.

Chapter 8 takes a closer look at the effect of data placement on system performance of a parallel database system. The data placement process can be divided into three phases, namely data partitioning, data distribution and data reorganisation. Several data partitioning and data distribution strategies are discussed. Among these, some have been implemented in commercial database systems while others have been included in prototype database machines. However, little work has been done to compare the relative performance of a parallel database system when different data placement strategies are applied to it. For this reason, an experiment has been conducted on the dual-level data placement to study the effect of such placement on the performance of a parallel DBMS.

Chapter 9 concludes the discussion of the thesis and presents future plans.

1.2 Authorship

Some of the material discussed in this thesis has been used for publication purposes. The following two papers have been published or submitted for publication. :

1. E. Dempster, N. Tomov, J. Lu, C. Pua, M.H. Williams, A. Burger, H. Taylor and P. Broughton, "Verifying a Performance Estimator for Parallel DBMS", in D. Pritchard and J. Reeve eds., *4th Int. Euro-Par Conf. on Parallel Processing*, LNCS, vol. 1470, Springer Verlag, Southampton, UK, pp. 126 - 135, September 1998.

2. C.S. Pua and M.H. Williams, "Modelling Parallel Databases With Process Algebra", submitted for publication to Parallel Computing, July 1998.

Meanwhile, a third paper is being prepared for submission to Distributed and Parallel Databases. A fourth paper will be considered in the near future.

Chapter 2

PARALLEL DATABASE SYSTEMS

2.0 Introduction

This chapter presents some of the background material to the thesis. Section 2.1 briefly discusses the situation that has led to the emergence of parallel database systems. In section 2.2, several computer architectures that are used as platforms for parallel computing are introduced. Section 2.3 briefly reviews the relation between data partitioning and transaction parallelism. Several techniques that can boost the I/O performance are also discussed. Section 2.4 reviews techniques that are used to maintain the consistency and reliability of database systems. Some examples of parallel database machines are described in section 2.5. The chapter concludes with a section describing some general parallel database systems.

2.1 Background

A database management system (DBMS) is a program designed to manage computerised records. For instance, a bank uses a DBMS to manage client transactions and records; a university uses a DBMS to manage student records; a hospital uses a DBMS to manage patient records and an individual may use a DBMS to manage personal records, and so on.

As DBMSs have developed, different approaches have been used to structure the data. Initially, two data models were introduced: the hierarchical data model and the network data model. The hierarchical data model places related data in a hierarchical tree structure; each node representing a collection of related records. The network data model is a generalisation of the

hierarchical data model, containing records and sets with syntax closely aligned with COBOL and with a low-level navigational interface. However, the queries against the data were difficult to generate because it required a substantial understanding of the complex structure of the data [86, 100].

Later, another data model has been introduced. Known as the relational model [27], it is commonly used in today's database systems. The collection of related data in this model is represented as a set of tables with rows and columns. It has the advantage that queries can be generated easily and quickly.

The conventional DBMS runs mainly on conventional computers. As the number of records increases, the database size grows accordingly. This results in increasing demands for more powerful machines that may be able to handle larger databases. However, mainframe computer designers have found it difficult to build machines powerful enough to meet the CPU and I/O demand serving a large number of users simultaneously or searching the huge databases [35].

Meanwhile, the speed of microprocessor CPU has increased significantly and their production costs are much lower than those of mainframe processors. However, given the state of technology, the growth in speed and the shrinkage in size of the microprocessor will eventually reach to a limit [90]. Furthermore, the cost to design an entire new and powerful microprocessor is extremely high. Consequently, computer manufacturers and researchers have begun to realise that it is more economical to put together several of the standard microprocessors in a computer than designing an entire new microprocessor for the desired power. This has led to the production of multiprocessor computers [73, 90]. These machines provide more power in total than the mainframe computers at a lower price [35].

A multiprocessor computer enhanced with the parallel programming technique [73, 88], allows multiple processes to be run simultaneously and hence increases the computing power [106]. While special purpose mainframe computers are becoming less common, database researchers begin to exploit the power of multiprocessor computers by combining the parallel

processing ability of the computer with the database management system. The resulting system, a parallel database management system, exploits the multiprocessor computer architecture in order to build a high performance and high availability database server at a lower price than an equivalent mainframe computer [35, 106]. It has been predicted in the future that high performance parallel database systems will displace the conventional centralised mainframe computer for transaction processing [35]. Despite the potential benefits of parallel database systems, their uptake has been slower than expected [30].

2.2 Parallel Computer Architectures

In general, according to Stonebraker [98] multiprocessor computer architectures can be categorised into three groups. They are known as i) *shared memory*, ii) *shared disks* and iii) *shared nothing*.

2.2.1 Shared Memory

In the shared memory architecture, all processors share direct access to a global memory and to all disks through a high speed interconnect. In other words, they share everything. Since the common memory and all disks are connected through a high speed interconnect, the problem of distributing data amongst the disks to improve performance is less crucial because every processor has equal access to all data. This aspect of load balancing is thus removed. Another advantage of this architecture is that the concurrency control is quite simple as every processor shares the same control information.

However, the interconnect between the processors and the common memory can be very complicated because every processor has a link to every memory module or disk. The network

traffic is also very busy because the processors are moving data and requests along the interconnect. As more processors are added, more will be accessing the shared memory and consequently introduce more interference or conflict-access to the shared resources [35, 106]. The degree of complexity and the traffic of the interconnect will also increase. This has limited the scalability of the computer architecture to tens of processor [106].

Because the memory space is shared by every processor, a memory fault can affect most of these processors thereby hurting the data availability of the system [106]. Examples of shared memory parallel database systems include DBS3 [6], Volcano [48] and XPRS [99], as well as HDM [84] and SiDBM [72].

2.2.2 Shared Disks

In the shared-disks architecture, each processor has its own exclusive private memory but has direct access to all disks through a high speed interconnect. Each processor can access the data on the shared disks and copy them into its own memory. In order to maintain data consistency, global locking and concurrency control protocols are essential.

This architecture has a significantly less complicated interconnect compared to the shared memory architecture. Load balancing is good because the data are stored on the disks as a whole. Given enough memory space for each processor, interference can be isolated from other processors and therefore increase the data availability [106].

However, this architecture requires a complicated distributed locking and concurrency control protocol. When a processor needs to update a data page, it has to ensure that no other processors have access to the data page. Once it obtains exclusive access and sets the appropriate locks, it copies the page into its memory and performs data manipulation. Then, it has to write the updated copy to the disks and make sure that every processor in the system will have access to the latest copy of the data page. This will create heavy traffic on the shared disks interconnect.

Examples of shared-disk parallel database systems include IBM's IMS/VS Data Sharing product and DEC's VAX DBMS.

2.2.3 Shared Nothing

In the shared nothing architecture, neither memory nor disk is shared among the processors. Such a system consists of multiple homogenous processing elements (PEs), each with its own exclusive memory and disk units. The PEs communicate with each other by sending messages via a high speed interconnect.

The shared nothing architecture has a fairly simple interconnect compared to the shared memory and shared disks architectures. Where the latter two architectures move large amounts of data across the interconnect, the shared nothing architecture moves only queries and filtered data. All accesses are performed locally and thus the traffic in the interconnect is minimised. This allows the shared nothing architecture to be scaled up easily to relatively large numbers of processing element (hundreds and probably thousands) [35]. With a proper data placement across the PEs, linear speed up and linear scale up could be achieved for a simple workload. By replicating the data among the PEs, high availability can also be achieved [35, 106].

On the other hand, this architecture requires a highly complicated locking and concurrency control protocol especially when the number of PEs is large. Load balancing is also difficult to achieve among the PEs and this relies on the effectiveness of the data placement strategies. The database systems that are based on this architecture include Bubba [14], EDS [2, 108], Gamma [32, 34], Teradata's DBC [90] and Tandem's NonStop SQL [101].

2.2.4 Architecture Comparisons

Comparing the three parallel computer architectures; shared memory provides a better performance for a smaller configuration (i.e. when the number of processors is small, for instance 20) because of the excellent load balancing [106]. On the other hand, shared disks and shared nothing demonstrate a better extensibility and availability than shared memory. On top of that, the shared nothing architecture can be easily scaled up to a higher number of processors than the other two [35].

Stonebraker [98] however has argued that with the existence of load balancing aids, load balancing in the shared nothing architecture should not be a serious problem. He has also claimed that deadlock is a rare occurrence in current systems and hence concurrency control is also not a serious concern in a well designed database system and therefore the shared nothing architecture should be the choice for today's parallel database systems [98].

While some researchers believe that the shared nothing architecture is the best choice among the three, others maintain that this is still an open issue and depends on various factors [9, 106]. Meanwhile, some researchers have begun to work on hybrid parallel computer architectures. For example, given the limited extensibility of shared memory and the load balancing problem of shared nothing, a hybrid parallel computer could have a shared nothing system in which each of its nodes is itself a shared memory multiprocessor. Here, the load balancing problem of a shared nothing machine can be simplified while maintaining the extensibility of a shared nothing system. An example of such a system can be seen in Teradata's P90 supercomputer database machine [18].

2.3 Data Partitioning and Transaction Parallelism

In a shared nothing parallel database system, a proper data placement is essential for load balancing. Ideally, every processing element can work simultaneously on independent datasets with little interference from other processing elements. This will allow a number of transaction requests to execute simultaneously. Furthermore, the parallelism inherent in a data-intensive application workload can be further exploited by transaction parallelism, which can enhance the system performance and availability [13]. The transaction parallelism can be divided into inter-transaction and intra-transaction parallelism. Inter-transaction parallelism enables the parallel execution of multiple transaction requests whereas intra-transaction parallelism allows the parallel execution of multiple queries or operations within a transaction request. In addition to this, the inter-operation parallelism which allows the parallel processing of a single operational command on different data streams across several processors can also be achieved [76, 106]. For this reason, relations are usually partitioned and distributed across the processing elements.

Data partitioning partitions a relation by dividing the set of tuples into a number of fragments either horizontally or vertically. Horizontal partitioning partitions a relation at the tuple level so that each fragment contains a subset of tuples of the relation. In vertical partitioning, each fragment contains a subset of attributes of the relation including the primary key of the relation. The simplest data partitioning strategy distributes the tuples among the fragments in a round-robin fashion. The *hash* partitioning strategy allocates tuples to fragments according to a hash function applied to the key attribute of each tuple whereas the *range* partitioning strategy clusters tuples with similar attribute values together in the same fragment.

These fragments are then distributed amongst the PEs according to a data placement strategy. For instance, the *round-robin* data placement strategy distributes fragments in a round-robin fashion. The *hash* data placement strategy places the fragments according to a hash function applied to the key attribute of the fragments. The fragments can also be distributed to the PEs according to size and access frequency [29].

Query processing in a database machine tends to be I/O bound because of the high disk access time compared to main memory access time [35, 71, 106]. Other than data partitioning, researchers have also been working on techniques that can boost the I/O performance of database systems. Among these techniques are disk array or disk striping [95, 23, 109, 49, 83, 91], and disk shadowing or disk mirroring [10, 11]. A disk array consists of a large number of small, inexpensive disks and a high bandwidth interconnect. It can provide very high throughput by exploiting the I/O parallelism, that is by 'striping' the file or database across the disks in the array, reading or writing to disk can be done in parallel and thus reduce the data transfer time [68]. Disk shadowing is a technique that is used to enhance the availability and reliability of the disks. It consists of a set of two or more identical disk images on different disks known as mirrored disks or shadow set. Here, data is duplicated and stored on the mirrored disks or shadow set. These techniques have been incorporated in XPRS [99] and Teradata's P90 [18].

2.4 System Consistency and Reliability

System consistency and reliability are equally as important in a parallel database system as are high performance and high availability. This can be achieved through a proper concurrency control protocol. The purpose of concurrency control is to maintain the consistency of the database and at the same time maintain the concurrency in the system [89].

The most common concurrency control mechanism is locking. This can be subdivided into centralised locking and decentralised locking. In centralised locking, locking information is control by a single node. In contrast, locking information in decentralised locking is distributed amongst the nodes.

The *two-phase locking* protocol is one of the basic locking algorithms. It states that no transaction should request a lock after it releases one of its locks [89]. There are two phases in

this protocol. In the first phase (*growing phase*), the transaction obtains locks and accesses the data. In the second phase (*shrinking phase*), the transaction releases its locks one after another until it terminates. This protocol can increase the degree of concurrency of the system by allowing other transactions to lock the data as soon as a transaction has released its lock(s) on the data. However, this is not easy to implement because the transaction manager has to know exactly when a transaction has obtained all of its locks. Besides, if the transaction decides to abort the operation just after it releases some of its locks, other transactions that have just obtained these locks have to abort their operations as well. Another protocol is known as the *strict two-phase locking* [86, 89]. The strict two-phase locking has a growing phase that is similar to the two-phase locking protocol. However, the strict two-phase locking protocol releases all locks simultaneously when the transaction terminates.

A transaction lock can be a shared lock or an exclusive lock. When a transaction only intends to read a data, it requests a shared lock for the data so that other transactions can also share access to the data. However, when a transaction intends to update a data, it requests an exclusive lock. No other transaction can access this data until the transaction has released the exclusive lock. A lock can be applied to a row, a page, the entire relation or even the entire database [63, 65].

When a transaction is executed on more than one site, the system needs a proper protocol to govern the consistency and reliability of the database. The *two-phase commit* protocol is commonly used for this purpose. There are two phases in this protocol. In the first phase (*voting phase*), the coordinating site or the coordinator prepares to commit. It sends a message to every participant and waits for the replies. The participants, after receiving the message from the coordinator, vote to either commit or to abort the transaction and return replies to the coordinator. In the second phase, the coordinator receives votes from the participants and then makes a decision based on the votes. If there is any vote for abort, the coordinator will decide to abort the transaction. It will send an abort message to every participant and prepare to abort the transaction. The participants will abort their transactions and send acknowledgements to the

coordinator. The coordinator will terminate the transaction and write a record to the log after receiving acknowledgements from all of the participants. If there is no vote to abort, the coordinator will send a commit message to all of the participants and enter the commit-state while waiting for the acknowledgements from all of the participants. The participants will commit their transactions and return acknowledgements to the coordinator so that the coordinator can write a record to the log and terminate (commit) the transaction.

A system may crash due to certain circumstances such as power failure. When this happens, the system has to recall all the unfinished transactions in order to maintain the consistency of the database as soon as it recovers. To achieve this, the system has to refer to certain records in order to perform the exact same operations. For this reason, a record (log entry) is written to a memory area (log buffer) as soon as a database operation takes place. This information will be used to remind the system to perform the same operations again or reverse the operations when necessary. Ideally, the log buffer will be written out to disk or to the sequential log file at an appropriate time so that this process will not introduce any extra cost to the overall system performance. Some systems implement a dual-buffer logging protocol, in which there are two log buffers that are used alternately during the logging process. When one of the log buffers is written out to disk, the other one is always available for log entries.

2.5 Database Machines

Since the first workshop on High Performance Transaction Systems in which Stonebraker presented the idea of multiprocessor computer architectures, numerous database machines based on these platforms have been produced. A few parallel database systems are discussed by way of illustration.

2.5.1 Bubba Prototype Database System

Bubba is a highly parallel computer system for data intensive applications that is based on a shared-nothing architecture [14]. The objective of the prototype system is to provide a better cost-performance compared with the conventional mainframe-based database systems of the early 90s. Bubba is implemented on a Flex/32 multiprocessor platform that consists of 40 nodes; each has a disk attached to it. The nodes communicate with each other through a shared common memory via a hierarchy of 32-bit buses. However, the common memory is partitioned in such a way that each node has an exclusive buffer pool of outgoing messages, which are managed exclusively by the node. These nodes can be divided into three types: Interface Processors (IP) for communicating with external host processors and coordinating query execution, Intelligent Repositories (IR) for data storage and transaction query processing, and Checkpoint-and-Log IR's (CIR's) for log maintenance. The data is partitioned across the nodes according to a hash or range data partitioning strategy. Queries are executed on the nodes that contain the relevant data in order to exploit the parallelism within individual transactions as well as multiple transactions. Bubba operates on a customised operating system called BOS.

2.5.2 GAMMA Database Machine

Gamma is a shared nothing relational database machine operating on an Intel iPSC/2 hypercube with 32 nodes and a disk attached to each node [34]. The prototype system focuses on the execution of select and join operations. The system consists of four processes: Catalog Manager, Query Manager, Scheduler Processes and Operator Processes. Catalog manager maintains the consistency of conceptual and schema information of each database among the copies cached by each user. Query Manager provides interface for ad-hoc queries, query parsing, query optimisation and compilation. Scheduler Processes control multi-site query and activate operator processes, as well as to prevent processors become bottlenecks.

The machine supports horizontal partitioning strategies such as round robin, hash, range and hybrid-range partitioning strategy [44]. As a query is being optimised, the partitioning information for each relation in the query is incorporated into the query plan. The Gamma machine employs a hash-based parallel algorithm to implement the complex relational operators such as join and aggregation function. The system operates on a customised operating system called NOSE. It implements two-phase locking protocol and each node has its own lock manager. A centralised deadlock detection algorithm is used to avoid lock conflicts in the system. Despite the single-user workload and hard-coded query, near linear speedup and scaleup for relational queries has been measured [33].

2.5.3 Teradata DBC

NCR/Teradata DBC [90] is a commercial relational database machine that is based on a shared-nothing parallel processing architecture. It is a highly parallel SQL system that utilises standard microprocessors, disks and memory. Teradata systems act as SQL servers to client programs running on conventional computers such as mainframes and PCs. Basically, the Teradata system consists of three types of processor. The Access Module Processor (AMP) is the database engine that manages the database rows held in its associated disk storage unit. The Interface Process (IFP) and Communication Processor (COP) manage the flow of requests and results between the host and the AMP. Y-net is the interconnect that links the AMP to the IFP and COP. Relations are hash-partitioned and hash-allocated across the subnets of the AMP. The system has demonstrated a near linear speedup and scaleup on relational queries and far exceeds the speeds of conventional mainframe computers to process large databases [36].

2.5.4 XPRS

XPRS is a high performance database system that runs on general operating system [99]. It is built on a shared memory architecture in the belief that shared-memory multiprocessors of several hundred MIPS would be available in the early 90s. In order to achieve high availability, it uses mirrored disk and distributed DBMS multi-copy technique. It is also assumed that when there is a software error, the operating system and data manager will recover instantly and thus increase the availability. XPRS has an added advantage of load balancing because the main memory and CPUs are automatically shared. In terms of data allocation, XPRS prefers a two-dimensional file system (FTD) rather than horizontal or vertical data allocation. It has been argued that in the latter strategy, when a disk is added or dropped, the data on the system would have to be reorganised and may be unavailable during the process. By striping the data, this problem can be resolved. XPRS has demonstrated briefly a near-linear speedup in its system evaluation using the two-phase optimisation of query execution plans when implementing on a shared memory system that consists of 12 processors and 5 disks [61].

2.6 General Parallel Database Systems

Besides the database systems that run on particular platforms, there are also commercialised general purpose parallel database systems, some of these are discussed below.

2.6.1 Informix XPS DBMS

Informix XPS DBMS [63] is a commercial product that utilises both inter-transaction and intra-transaction parallelism. It also utilises intra-operator parallelism by dividing query trees into subtrees, which are then sent to different PEs for execution [113]. Informix XPS DMBS takes a

shared-nothing approach to managing data to minimise operating system overhead and reduce the traffic in the interconnect.

In the Informix DBMS, each PE runs its own instance of the database (co-server) which consist of basic Online XPS services such as logging, recovery, locking and buffer management. Each co-server owns a set of disks and the partitions of the database reside on the disks. The co-servers may interact and co-ordinate activities with each other.

Each co-server has a request manager, a query manager, a query optimiser, a metadata manager and a scheduler. The request manager decides how a query should be divided and distributed while ensuring load balancing across the PEs. The query optimiser determines the best way to execute a request. The metadata manager determines where the data resides, whereas the scheduler distributes the requests across the PEs.

In Informix DBMS, tables are partitioned into fragments that are distributed amongst the nodes. Three partitioning strategies are provided by the system: system defined hash, round-robin and expressed-based (user defined).

The Informix XPS DBMS uses the principle that the data operations are always executed where the data resides. For this reason locking is managed locally at each co-server. Transactions that are running on the same node share access to data in the local memory. The system uses mechanisms called latches and locks to coordinate the transactions and prevent simultaneous writing of the same data. The Informix DBMS uses shared and exclusive locks to achieve this. Locks are applied at the page level only.

Writing a buffer back to the disk is managed by a background process (page-cleaner thread). This process is only required when (i) there are too many dirty pages; (ii) the cache is full; or (iii) during a checkpoint. In addition to this, Informix DBMS uses a dual-log buffer to ensure that when one of the log-buffers is being flushed to the disk, the other is always available for logging. The flushing of the log-buffer is also done in the background.

2.6.2 INGRES Cluster DBMS

INGRES is a general purpose concurrent relational database management system that allows multiple users to access the same data at the same time [64, 65]. The Goldrush Ingres Cluster DBMS is a parallel version of INGRES that is based on the CA-OpenIngres 1.1 VAX Cluster model [113]. The Ingres Cluster DBMS consists of one or more DBMS servers that can execute multiple transactions simultaneously, one or more Communication servers, a Recovery process, an Archive process and a Node Support process. The Communication servers provide access points to the DBMS from the external network. The Archiver process copies log records from the log files to the journal files for each database. The Node Support process acts as a watchdog that reacts to process failures and other error conditions. In the Ingres Cluster DBMS, disk I/O to different nodes is coordinated by a distributed file system. A distributed lock manager is used to coordinate the locks on the data.

The Ingres Cluster DBMS supports two types of locks. The *logical lock* follows the strict two-phase locking protocol where the locks are held for the life of a transaction until the transaction terminates. The *physical lock* is used to synchronise the access to the resources. It is granted and released within a transaction. A logical lock can be in several modes: *exclusive*, *shared*, *intended exclusive*, *intended shared*, *shared intended exclusive* or *null*. It can be applied to a page, a table or the entire database. The Ingres Cluster DBMS locking system controls locking by managing and queueing the lock requests as well as detecting the deadlock situation among the requests with the help of the *distributed lock manager* on each processing element.

The Ingres Cluster DBMS also consists of a shared memory logging database, which is used to keep track of every transaction that takes place. This record will be used for system recovery after system failure. The system supports *group commit* protocol in order to minimise the frequency of disk I/O operations. Checkpoints are usually performed in an exclusive time slot.

2.6.3 Oracle 7 Parallel Server

Oracle supports three forms of parallel processing systems: shared memory, shared disks and shared nothing [87]. However, only the Oracle 7 Parallel Server that supports the shared nothing system is discussed.

Oracle 7 Parallel Server utilises inter-transaction parallelism although its parallel query option also supports intra-query parallelism (including inter- and intra-operator parallelism) under certain circumstances. It consists of multiple instances. Each instance has its own system global area (SGA) and a set of redo log entries. Each instance also contains a parallel cache management (PCM) locks area, background processes and user processes, and files, including data files, control files and redo log files, in its SGA. The background processes include DBWR, LGWR, ARCH, PMON, SMON, CKPT, RECO and LCKn. The DBWR manages the database buffer cache, whereas the LGWR writes the information in the redo log buffer to disk. The ARCH copies the log files to the disk or tape. The PMON (Process Monitor) and SMON (System Monitor) reclaim database resources that are no longer needed. The CKPT updates log file headers during checkpoints. The RECO resolves failed distributed transactions, and LCKn is used to control inter-instance locking. All instances in the parallel server share the same set of data files and control files. However, every instance has its own log entries and archived logs, which must be accessible to other instances in case of instance failure and media recovery.

In Oracle, data sharing is controlled by parallel cache management by using PCM locks. A data block can be present in several SGAs at the same time. The PCM locks ensure that the database cache is kept consistent for all instances at all time. The background lock process (LCK) manages the locks used by an instance and coordinates the request for those locks by other instances. It uses the distributed lock manager (DLM) to coordinate the buffer cache of the different SGAs in a parallel server. Oracle supports row, block, and table level locking. Locks can be shared or exclusive.

Within a single instance, Oracle uses a buffer cache in memory to reduce the amount of disk I/O necessary for database operations. The system only reads data blocks from the disks if they are not already in the buffer. The writing back is deferred until there is not enough space for new data, during a checkpoint or when another instance needs the data blocks.

Chapter 3

PERFORMANCE EVALUATION AND PREDICTION

3.0 Introduction

High performance is a key objective of parallel database systems. To achieve this, it is necessary to measure the performance of such systems in order to show that the systems satisfy the requirement. This chapter discusses the general concept of evaluating and predicting the system performance of parallel database systems. Section 3.1 presents the general performance metrics and evaluation techniques. Section 3.2 describes the benchmarks that are used to test the performance of database systems. Section 3.3 encloses some examples of performance evaluation whereas section 3.4 reports on some works related to performance prediction in parallel database systems.

3.1 Performance Metrics and Evaluation Techniques

There is no single metric that can be used to measure the performance of all applications of a computer system. The performance metrics depend on the application domain [67]. Each system is typically designed to solve certain problem domains and is incapable of performing other tasks [50]. Nevertheless, the performance metrics can be broadly categorised into responsiveness and usage level. In the responsiveness category, the measures are intended to evaluate the speed of a given task accomplished by the system. The metrics can be *waiting time*, *processing time*, *queue length* and so on. The usage level category is intended to measure the throughput and utilisation of various resources of the system.

There are three basic approaches to evaluate a system's performance [67, 69]. The first approach is measurement or benchmarking. According to this approach, the measurements are taken while the system is running. The performance estimated in this case is always accurate. However, this approach is very costly as it may entail the acquisition of new equipment simply for the evaluation purposes.

The second approach is simulation, in which case a model of the system is built to reflect the behaviour of the system. The simulation model is evaluated against a set of parameters. The parameters are changed and the simulation model is rerun. The advantage of this approach is that the detailed features of the system can be included into the model. However, if too many detailed features are included, the simulation model can become more complicated and require more running time and hence increase the cost of the performance study. This approach is also very time consuming in producing a reliable set of results.

The third approach is analytical modelling, in which case a mathematical model of the system, or part of the system, is constructed. The advantage of this approach is that it provides a good insight into the working of the system. A simple analytic model can be solved easily and yet produces accurate results [67].

3.2 The Benchmarks

A benchmark is a set of specifications that is used to test system performance. They have become increasingly important as more people depend on them to compare and evaluate the performance of computer systems. There was no standard benchmark two decades ago. Many vendors created their own ad hoc benchmark so that they could evaluate their system performance against the competition.

The first standard benchmark was known as the Wisconsin Benchmark, which was developed at University of Wisconsin in the early 80s. Thereafter, several standard benchmarks have been developed. Among those that are commonly used to evaluate the performance of database systems are TPC-A, TPC-B and TPC-C of Transaction Performance Council, and AS^{3}AP of ANSI.

3.2.1 TPC-A and TPC-B Benchmark[1]

The TPC Benchmark™ A is designed to test OLTP(OnLine Transactions Processing) systems [97]. It emphasises update-intensive database services. The metrics used in this benchmark are throughput, which is measured in transactions per second (tps) and also the associated price per tps. The TPC Benchmark™ A requires multiple on-line terminal sessions and can be run in wide area or local area network configurations.

The TPC Benchmark™ B [103] is not for OLTP systems and therefore it does not require on-line terminals or networks. It is used to test transaction processing systems that emphasise update-intensive database services. The metrics used for this benchmark are throughput (measured in transactions per second (TPS)) and the associated price per TPS.

Both TPC-A and TPC-B are described in terms of a hypothetical bank that has several branches. Each branch has multiple tellers. The bank has many customers with accounts. The database has three base relations (*Branch*, *Teller* and *Account*) and a history of recent transactions processed by the bank (*History*). A transaction takes place when a customer makes a deposit or withdrawal at a particular branch through one of the tellers. The transaction profile therefore consists of three updates (for relations *Branch*, *Teller* and *Account*) and one insert (for relation *History*).

[1] Both TPC-A and TPC-B benchmarks are now obsolete.

3.2.2 TPC-C Benchmark

TPC Benchmark™ C [93, 104] is the third approved benchmark of Transaction Processing Performance Council (TPC). It is also an OLTP benchmark, but it is different and more complicated than TPC-A. It has multiple transaction types and more databases. It also has different execution structures.

TPC-C simulates an environment in which terminal operators execute transactions against a database. It is composed of basic operations designed to exercise the system functionalities in a complex OLTP scenario. TPC-C is a mixture of read-only and update-intensive transactions that describe the activities of a wholesale supplier. These activities include entering and delivering orders, recording payments, checking the status of an order and monitoring stock levels at the local warehouse. The relations involved in this study are *Order_line*, *History*, *Customer*, *Order*, *Stock*, *Parts*, *Item*, *New_order*, *District* and *Warehouse*. The most frequent transaction consists of an order entry and a customer payment receipt. Other relatively less frequent transactions include order status checking and local warehouse stock level examination.

3.2.3 Wisconsin Benchmark

This was the first standard database benchmark developed [12, 31]. The initial purpose of this benchmark was to test the performance of the major components of a relational database system. The benchmark consists of three relations; one with 1000 tuples and two others each with 10000 tuples. The query suite is designed to test the performance of all basic relational database operations such as selections, projections, single and multiple joins, aggregate functions, insertion, deletions and updates. In addition to this, most queries are tested twice: the first time using a clustered index, and the second time not using one.

The original benchmark was originally single-user in nature and did not perform batch operations. Nevertheless, extension to the benchmark has been made and it is now fairly extensively used to evaluate database systems running on parallel processors. It has been used to measure the speed-up and scale-up of parallel database systems.

3.2.4 AS^{3}AP Benchmark

The AS^{3}AP (ANSI SQL Standard Scaleable Portable) benchmark is designed to be used to test and compare relational database systems with vastly different architectures and capabilities over various workloads [105]. The basic performance metric is the *equivalent database ratio*, which is the maximum size of the AS^{3}AP database for which the system is able to perform the designated AS^{3}AP set of single and multi-user tests in under 12 hours.

The AS^{3}AP tests are divided into two modules: single user tests and multi-user tests. The single user tests include the utilities for loading and structuring the database and the queries designed to test access method and basic query optimisation. The multi-user tests concentrate on the OLTP workloads, information retrieval workloads and the mixed workloads of short and long transactions.

The only measurement required by AS^{3}AP is the query elapsed time within the 12 hour limit. The AS^{3}AP consists of four main relations and a tiny relation, which has only one column and one tuple, and is used to measure data loading overhead. The four main relations have the same average tuple width and the same number of tuples. Each relation has 10 attributes and the average size of each tuple is 100 bytes. The size of the relation can be scaled from 1 megabyte to 100 gigabyte by varying the number of tuples. Although the four relations have the same number of attributes and average tuple size, each one has its own unique characteristics. For instance in the first relation (*uniques*), all the attributes have unique values. Whereas in the second relation (*hundred*), most of the attributes have exactly 100 unique values and are correlated. In the third

relation (*tenpct*), most of the attributes have, at the most, 10 percent unique values and the forth one (*update*) is customised for update purposes.

3.3 Examples of Performance Evaluation

This section discusses how a state-of-the-art machine is performance tested. The discussion concentrates on the workload that was used during the test.

3.3.1 Gamma Performance Evaluation

The Gamma machine was tested with the standard Wisconsin Benchmark [34]. Three metrics were studied, namely; response time, speed-up and scale-up. In the test, the three relations stated in Wisconsin Benchmark were configured to have 100000, 1 million and 10 million tuples respectively. Each tuple of each relation is 208 bytes in size.

For response time results, the test was designed to determine how the system would respond as the size of the relations was increased while the number of processors was kept at 30. For speed-up experiments, the number of processors was varied from 1 to 30 while the size of the test relations was fixed at 1 million tuples. Whereas for the scale-up experiments, the number of processors was varied from 5 to 30 while the test relations were increased from 1 to 6 million tuples. The relations were partitioned using a hash partitioning strategy. The page size was configured to be 8 Kbytes. The system was then tested for the basic database operations or queries such as selection queries, join queries, aggregation queries and update queries. The speed up results for both selection and join queries were claimed to be almost perfectly linear while the scale-up results were encouraging [34].

3.3.2 Bubba Performance Evaluation

Three metrics were adopted in Bubba's performance evaluation. They were OLTP throughput scale-up, batch scale-up and batch speed-up. An order-entry system application or workloads has been developed to test the prototype system performance [14]. The workload was composed of five order-entry transactions: New-Order, Order-Shipped, Payment, Suggested-Order and Store-Layout. For the workload, the database consisted of eight relations.

The OLTP throughput scale-up test was used to measure the throughput of small transactions as the number of nodes (IRs) and the database size were increased. The goal was to increase the transaction throughput in proportion to the relative increase in system size, while maintaining the same transaction response times. The batch scale-up test was used to measure the response time of the batch-program or decision support queries as the number of nodes (IRs) and the database size were increased. The goal was similar to the OLTP throughput scale-up test. The batch speed-up test was used to measure the improvement of large query response times as the number of nodes were increased, while the relations size were kept constant.

3.4 PDBMS Performance Prediction Related Work

With the increasing interest in using parallel computers as database servers, there is a corresponding interest in performance prediction of parallel database systems. Both analytical and simulation approaches have been used and reported in the literature.

3.4.1 Discrete Simulation System

Marek et al [76] have developed a comprehensive discrete simulation system to study the performance of a shared-nothing parallel database system. The model supports three different

workload types for performance evaluation: the debit-credit workloads (TPC-A), a synthetically generated relational query and a real life workload. The performance metrics for the study are system response time and system throughput.

The simulation system consists of two components, the workload generator and the processing subsystem. The workload generator generates a stream of transactions for the processing elements (PEs) where the actual transaction processing takes place. The model employs a horizontal data partitioning strategy and a distribution table that defines for every partition P_j and processing element PE_i which fragment of P_j is allocated to PE_i.

Each PE consists of a transaction manager, a concurrency control component, a buffer manager, a communication manager and a CPU server. The transaction manager employs the two-phase commit protocol while the concurrency control component follows the distributed strict two-phase locking. It is assumed that the smaller relations (such as Branch, Teller and History of TPC-A) are memory-resident. The larger relations may be memory-resident or stored on disk. The buffer manager uses the LRU replacement strategy to manage the buffer.

The simulation focuses on the influence of inter- and intra- operation parallelism and scalability. For this reason, the distribution and parallelism of operation statements are already determined at load generation time.

The simulation results have been compared to those predicted by the EDS prototype system and are claimed to be realistic in their paper.

3.4.2 SMART

SMART (Simulation and Model of Application based on Relational Technology) is a performance simulation tool for relational and transactional applications [15, 37]. The simulation technique of SMART is based on a model of queueing networks. SMART consists of four main

components: *SmartForm*, *SmartEngine*, *SmartDictionary* and *SmartEvaluator*. SmartForm is used as the user interface. SmartEngine acts as the simulator motor. SmartDictionary is the database that contains the base information in SMART, and SmartEvaluator is used to estimate the SQL query I/O costs from the SQL query text and the evaluation of each operation in the query.

When simulating an RDBMS model, the specification of the RDBMS has to be incorporated into the design of SMART. The model will contain the important algorithms of the RDBMS such as cache management, relational operations, concurrency control, and so on. This model will be developed in a simulation language which supports the definition of time constrains, the execution and the synchronisation of several tasks in parallel. Next, the behaviour of the RDBMS is quantified. A set of measures is collected as the basic costs (e.g. locking time, data sorting time, and so on). The numbers and some extrapolation laws are then compared with the simulation model. The RDBMS model is then validated for the machine on which the measures have been taken.

SMART has been used to model the performance of Oracle 7 Parallel Server running on Goldrush [16]. In the study, the assessments are divided into four parts: Query assessment, Parallel query assessment, Multi user assessment and Parallel multi use assessment. The first two assessments use the AS^{3}AP benchmark and are used for the DBMS optimiser analysis, whereas the others use the TPC-C benchmark as the workload. The platform is configured to have six processing elements with four of them are attached to four disks each, while the other two are attached to two disks each.

3.4.3 Analytical Estimator

Zhou et al [111, 112] have designed an analytical tool called STEADY (System Throughput Estimator for Advanced Database sYstems) to estimate the performance of a shared nothing

parallel database system. The STEADY system estimates system throughput by analysing a set of parameters relating to the database application, the system architecture, the data placement strategy and the relations of a database. It consists of four major modules: DpTool, Profiler, Modeller and Evaluator.

The DpTool takes as input the detail of the relations, and information on the system architecture and queries, together with the data placement strategy in order to generate a data placement scheme. It consists of three sub-modules (declustering, placement and redistribution) to organise the data allocation. Various data placement algorithms are incorporated including hash, round-robin, size, heat, and so on. The data placement scheme generated will be used as input to the Modeller.

From the input data, the Profiler generates a statistical profile for the base relations and estimates the statistical profile for the temporary relations which are generated during query execution. The base relation profile is only assessed and modified by the temporary relation profiler whereas the temporary relation profile is required by the Modeller.

The Modeller takes the benchmark profile as input and analyses this against the profile information and data placement scheme in order to estimate the data operation costs and the I/O access costs. The I/O access costs include the costs of logging, locking and read/write access. The data operation costs refer to the CPU time consumed by non-I/O data operation processes. These costs are stored in a workload profile.

The Evaluator uses the workload profile to compute the average transaction on each PE and to record the consumption of each resource in the transaction running on each PE. It identifies the bottlenecks of the system and estimates the system throughput.

The STEADY system was originally based on the Ingres Cluster model but recently it has been extended to model the Informix XPS and Oracle Parallel Server [30]. Apart from the system throughput, it can now predict system response time given a transaction arrival rate. It has been

used in a range of experiments, especially in the study of the effect of data placement on the performance of a parallel database system [110, 112].

3.4.4 RDBMS Performance Evaluation

Lie et al [75] studied the performance of a heterogeneous multiprocessor relational database system through a queueing network model. Each processor or group of processors is specialised in certain tasks.

The queueing network model consists of six service stations. They are SEMM (secondary memory and its manager), RUP (data filter), MAMM (main memory and its manager), SOP (sort processor), COP (conversion processor) and IRP (inter-record processor). The SEMM uses a set of RUPs to allow data access by contents. The MAMM is the common memory (or central memory) of the multiprocessor system. The SOP is specially tailored to perform a 4-way merge sort while the COP connects the RDBMS data bus to the database supervisor. The IRP is specialised in performing database operations such as aggregation functions and inter-relation operations (e.g. join).

The model supports three synthetic workloads that reflect different database applications involving data retrieval, updates, inter-record and inter-relation operations. The three workloads are meant to give a good utilisation study of different aspects of the RDBMS.

Chapter 4

PROCESS ALGEBRAS AND PEPA

4.0 Introduction

This chapter presents the formalism that is used to model the performance of parallel database systems in this thesis. The formalism is known as PEPA (Performance Evaluation Process Algebra) which is an extension of the ideas of classical process algebras. Section 4.1 briefly discusses the background to process algebras. An example of classical process algebra, CCS, is also introduced. Section 4.2 presents PEPA in detail. The chapter concludes with works related to PEPA.

4.1 Background

Process algebras are mathematical theories that are used to model communication and concurrent systems. Examples of process algebras are Calculus of Communicating Systems (CCS) [78, 79, 40], Communicating Sequential Processes (CS) [59] and Algebra of Communicating Processes (ACP) [7]. In process algebras, a system is described by its components and the communications between these components.

In the classical or pure process algebras, time is abstracted away within a process. All processes are assumed to be instantaneous. The relative timing is represented in the traces of the processes. In order to model the real time behaviour of a system, timed extensions of process algebras, for example temporal CCS [81], have been introduced. Another problem with pure process algebras is that in order to model systems in which there is uncertainty about the system

behaviour, this too is abstracted away. This causes the choices between actions or behaviour to become non-deterministic. The introduction of probabilistic extensions of process algebra [66] allows the uncertainty to be quantified as non-deterministic choice and is replaced by a probabilistic choice.

4.1.1 Calculus of Communicating Systems

The Calculus of Communicating Systems (CCS) was introduced in the early 80s. It aimed to allow users to represent a real system with the terms or expressions of a general mathematical model, and to manipulate these terms in order to analyse the system behaviour.

This theory assumes that a system is composed of several parts, each acting concurrently and independently from other parts. Sometimes these parts may have to communicate with each other to achieve certain tasks. These parts are called *agents* and they engage in *actions*. Each action of an agent is either an interaction with its neighbouring agents or it occurs independently. A communication takes place when an agent performs a *handshake* with another agent. For this to happen, one of these agents must be an *active* agent while the other is a *passive* one.

CCS provides a small set of combinators that enable an agent to perform a sequential action, a choice, or a concurrent action. CCS follows a set of transition semantics or rules for each combinator to outline the actions that an agent can perform. From this a derivative graph can be constructed which is useful for reasoning about the agents and the systems they represent.

A prototype tool called Concurrency Workbench (CWB) [82] has been developed to support the development of CCS specifications. It allows a system to be defined in the syntax of CCS and performs various analyses of the system. For instance, it can estimate the number of reachability states and compare the observational equivalence of two models, as well as tracing through the transition states of the model in a controlled fashion.

4.2 Performance Evaluation Process Algebra

Performance Evaluation Process Algebra (PEPA) [53, 54, 55] is a stochastic extension of classical process algebra. It enables the modelling of stochastic processes in a structured fashion. This language has been developed to investigate the impact of the compositional features of process algebra upon performance modelling.

In PEPA, a system is expressed as an interaction of *components* that engage in *activities.* These components correspond to the identifiable parts in the system, or the roles in the behaviour of the system. Each component has a behaviour that is defined by the activities in which it can engage. Every activity has an action type and an associated duration (which is a random variable with an exponential distribution, and is represented by a real number parameter known as the activity rate). The activity is written as (α, r), where α is an action type and r is the activity rate. The associated duration tot he activity makes PEPA differ from classical process algebras such as CCS. This is necessary for performance evaluation.

4.2.1 Syntax and Semantics

PEPA also provides a small but powerful set of combinators. The combinators allow the expression of the behaviour of components to be constructed through the activities they engage in and the interaction between them. The syntax for terms in PEPA is defined as follows:

$$P ::= (\alpha,r).P \mid P \langle L \rangle Q \mid P + Q \mid P/L \mid X \mid A$$

1. **Prefix:** $(\alpha, r).P$

   This is the basic mechanism by which the behaviour of a component is constructed. The specification $(\alpha, r).P$ will carry out activity (α,r) which has action type α and a duration of mean $1/r$. After completion of this activity, the component will behave as component P.

It is assumed that there is always an implicit resource that facilitates the activities of the component and which is not modelled explicitly. The time elapses to complete such an activity represents the time consumed by the component when using the resource.

2. **Choice:** $P + Q$

The specification $P + Q$ represents a system which may behave either as P or as Q but not both P and Q at the same time. The first activity to complete distinguishes one of the components. The other component of the choice is discarded. The continuous nature of the probability distributions ensures that the probability of P and Q both completing an activity at the same time is zero. The choice combinator represents the competition between components over an implicit resource.

3. **Cooperation:** $P \langle L \rangle Q$

The set of action type L, the cooperation set, determines the interaction between components P and Q. P and Q may proceed independently and concurrently with any activity whose action type is not contained in set L. However, for any activity whose action type is contained in set L, components P and Q must cooperate or synchronise on this activity. These shared activities will only be enabled in $P \langle L \rangle Q$ when they are enabled in both P and Q; i.e. one component may be blocked waiting for the other component to be ready to participate. When P and Q cooperate over an activity, the activity rate of the cooperation will be the rate that reflects the slower participant.

A component may be passive with respect to an activity. In this case, the activity rate is left unspecified, denoted as (α, infty). This means that a component may be required to synchronise in order to achieve an activity of that type although the component does not contribute to the work involved.

If the set L is empty, both component P and Q proceed concurrently without any interaction.

4. **Hiding:** *P/L*

The component will behave as *P* except that any activity of type contained in set *L* is not visible to the external observers or components. They appear as unknown type and can be regarded as an internal delay by the component.

5. **Variable:** *X*

If *E* is an expression of a component that contains a variable *X*, then *E{P/X}* denotes the component formed when every occurrence of *X* in *E* is replaced by component *P*.

6. **Constant:** *A*

A constant is a component whose meaning is given by a defining equation $A = P$, which gives component *A* the behaviour of component *P*.

The operational semantics of PEPA are quite straightforward and are summarised in **Figure 4.1.** The first rule **(Prefix)** is straightforward. Component $(\alpha, r).P$, after performing activity (α, r), will become component *P*. There are two rules for **Choice.** The first one states that if component *P* has a transition via activity (α, r) to become P', then component $P + Q$ has the same transition to become P' via activity (α, r). The second rule says the same for component *Q* in $P + Q$. And so on.

An example of a PEPA specification for the Producer-Consumer model is as follows:

$Q_0 = (put,infty).Q_1;$
$Q_i = (put,infty).Q_{i+1} + (get,infty).Q_{i-1};$
$Q_n = (get,infty).Q_{n-1};$

$P = (produce, r_0).(put,r_1).P;$
$C = (get, r_2).(consume,r_3).C;$

$System = (P <> C) <put,get> Q_0;$

where $n \in I^+$, $i \in \{1..n-1\};$

Prefix: $$\frac{}{(\alpha, r).P \xrightarrow{(\alpha, r)} P}$$

Choice: $$\frac{P \xrightarrow{(\alpha, r_1)} P'}{P + Q \xrightarrow{(\alpha, r_1)} P'} \qquad \frac{Q \xrightarrow{(\beta, r_2)} Q'}{P + Q \xrightarrow{(\beta, r_2)} Q'}$$

Cooperation:

$$\frac{P \xrightarrow{(\alpha, r)} P', \text{ where } \alpha \notin \mathbf{L}}{P \langle \mathbf{L} \rangle Q \xrightarrow{(\alpha, r)} P' \langle \mathbf{L} \rangle Q} \qquad \frac{Q \xrightarrow{(\beta, r)} Q', \text{ where } \beta \notin \mathbf{L}}{P \langle \mathbf{L} \rangle Q \xrightarrow{(\beta, r)} P \langle \mathbf{L} \rangle Q'}$$

$$\frac{P \xrightarrow{(\alpha, r_1)} P', Q \xrightarrow{(\alpha, r_2)} Q'}{P \langle \mathbf{L} \rangle Q \xrightarrow{(\alpha, R)} P' \langle \mathbf{L} \rangle Q'} \quad \text{where } \alpha \in \mathbf{L}, \quad R = \min(r_1, r_2)$$

Hiding:

$$\frac{P \xrightarrow{(\alpha, r)} P', \text{ where } \alpha \notin \mathbf{L}}{P/\mathbf{L} \xrightarrow{(\alpha, r)} P'/\mathbf{L}} \qquad \frac{P \xrightarrow{(\alpha, r)} P', \text{ where } \alpha \in \mathbf{L}}{P/\mathbf{L} \xrightarrow{(\tau, r)} P'/\mathbf{L}}$$

Constant: $$\frac{P \xrightarrow{(\alpha, r)} P', \ A = P}{A \xrightarrow{(\alpha, r)} P'}$$

Figure 4.1. Operational semantics of PEPA.

4.2.2 Performance Evaluation

To evaluate a PEPA model, a derivative set is obtained from the model following the operational semantics of PEPA. A transition diagram or derivative graph of the system can be produced from the derivative set. A derivative set is a set of components that capture the reachable states of the system. A state, C', is reachable from component C if there exists a sequence of activities such that after performing such activities, component C may become component C'. A derivative graph is a multi-graph that has the component defining the model as its initial node. Each

subsequent component or derivative is a node in the graph. The arc between the nodes represents the possible transition between the corresponding components.

By associating each node with a state, the underlying stochastic process can be generated. The restriction of the activity rate to be exponentially distributed is to ensure that the underlying stochastic process is a continuous time Markov process[1] [1, 38, 42]. The infinitesimal generator matrix **Q** (or state transition rate matrix) can be formed by taking the transition rates $q(C_i, C_j)$ (i.e. the transition from component C_i to component C_j) as the off diagonal elements q_{ij}, and the negative sum of the transition rates as the diagonal elements q_{ii}. The equilibrium probability distribution Π can be computed by solving the equation $\Pi Q = 0$ subject to the normalisation condition $\sum \Pi(C_i) = 1$. A system is in an equilibrium state such that the probabilities for the system being in a particular state, has settled down and are not changing with time [69]. In order to achieve equilibrium, a PEPA model must be *finite* and *irreducible*. A PEPA model is finite if its derivative set contains a finite number of components. A component is irreducible if its behaviour is always repeated. Regardless of how the model evolves from this component, it will eventually return to this point and this set of behaviours.

The system performance measures can be obtained from the solution by using the notion of reward structures proposed in [53]. A reward structure is used to provide a general framework for specifying and deriving performance measures over continuous time Markov processes. In which case, rewards are associated with certain activities within the system. Performance measures, such as response time and throughput, are then derived from the total reward based on the equilibrium probability distribution. Forming the reward structures has been done manually and explicitly. Nevertheless, Clark et al [26] have been working on the PEPA reward language which will allow a modeller to specify the reward structures without having to identify the state manually.

---

[1] A stochastic process is a Continuous Time Markov Process provided that $P\{X(t_{n+1}) = x_{n+1} \mid X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, \ldots, X(t_0) = x_0\} = P\{X(t_{n+1}) = x_{n+1} \mid X(t_n) = X_n\}$, $t_{n+1} > t_n > .. > t_0$.

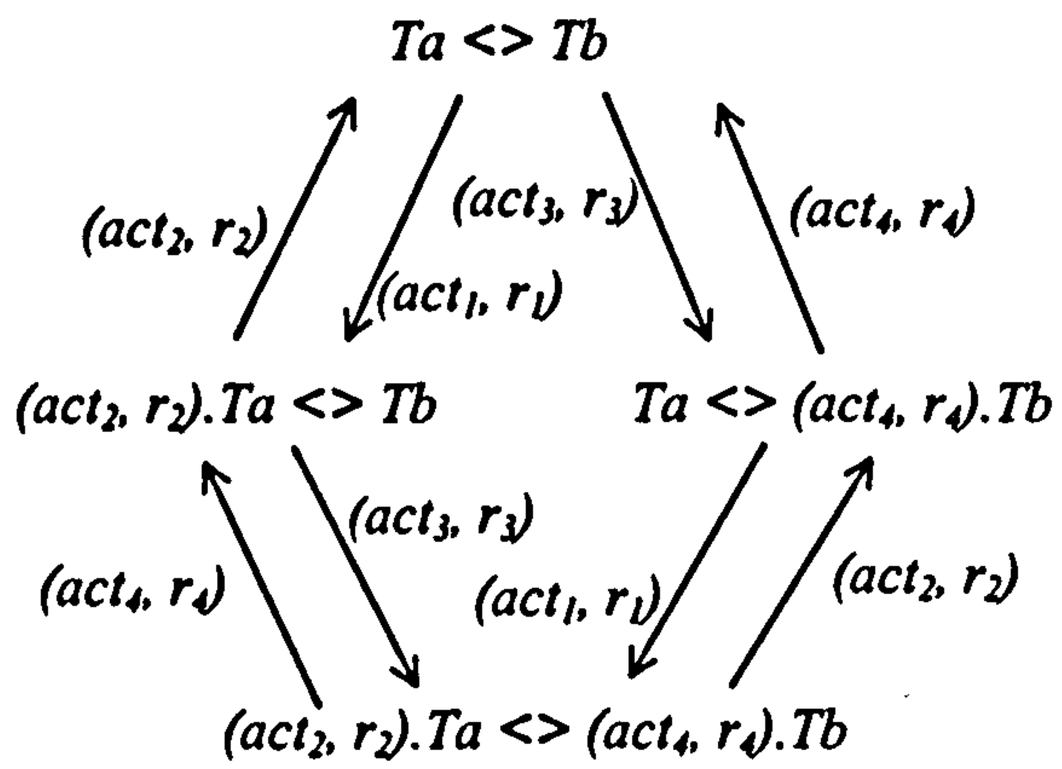
The PEPA workbench [45] is used to generate the infinitesimal generator matrix **Q** from the PEPA model. The matrix is stored in a file and the performance measures are then obtained by solving the set of linear equations with the help of a mathematical package.

Although various performance measures can be obtained from the PEPA model, only the system throughput is considered for the purpose of this study. The system throughput is measured in transactions per second subject to a residence time constraint. The performance measure can be considered as a rate-based measure that corresponds to the predicted rate at which some activity occurs. In this case, the reward is equal to the activity rate of the activity enabled. The throughput is the product of the rate of the activity, and the probability that the activity is enabled.

As an example, consider

$$\#T = Ta <> Tb;$$
$$\#Ta = (act_1, r_1).(act_2, r_2).Ta;$$
$$\#Tb = (act_3, r_3).(act_4, r_4).Tb;$$

The derivative graph of the above model will be



By associating each derivative with a state, the underlying stochastic process can be generated:

$$X_0 \leftrightarrow Ta <> Tb$$

$$X_1 \leftrightarrow (act_2, r_2).Ta <> Tb$$

$$X_2 \leftrightarrow Ta <> (act_4, r_4).Tb$$

$$X_3 \leftrightarrow (act_2, r_2).Ta <> (act_4, r_4).Tb$$

And the matrix **Q** looks like follows:

$$\mathbf{Q} = \begin{bmatrix} -r_1 - r_3 & r_1 & r_3 & 0 \\ r_2 & -r_2 - r_3 & 0 & r_3 \\ r_3 & 0 & -r_3 - r_1 & r_1 \\ 0 & r_4 & r_2 & -r_2 - r_4 \end{bmatrix}$$

The throughput of this example will be the number of activity (act_4, r_4) completed in a second. In this case, the reward associated with the states will be the activity rate(r_4) of activity (act_4, r_4). Thus, the throughput is:

$$T = (\Pi(X_2) + \Pi(X_3)).r_4$$

The equilibrium probability distribution $\Pi(X_i)$ can be computed by solving the linear equation system $\Pi Q = 0$ subject to the normalisation condition $\Sigma\, \Pi(X_i) = 1$.

4.2.3 Notions of Equivalence

PEPA models are prone to the problem of state space explosion when the complexity and the size of systems modelled increase. For this reason, notions of equivalence that aim to solve this problem are presented.

The notions of equivalence are the criteria that determine the similarity between two entities. There are three classes of equivalence: system-to-model, model-to-model and state-to-state. However, little formal development has been done on the former two notions of equivalence [53]. In contrast, there has been much work on the state-to-state equivalence as it forms the basis of the aggregation techniques for reducing the states of the underlying Markov process.

Hillston [53] presents four notions of state-to-state equivalence that may be used as the basis for a model simplification technique. They are isomorphism and weak isomorphism, strong bisimilarity and strong equivalence. However, only the compositional strong equivalence aggregation technique that replaces the cooperating components by strongly equivalent lumped

components is considered in this study. This is because the isomorphism is a very strong notion of equivalence such that it is too strong to be used as a model simplification technique. The weak-isomorphism may generate Markov processes that are not equivalent for both the weakly isomorphic component and the initial compact form component. Whereas the strong bisimilarity alone, is not sufficient to ensure that the aggregated components will exhibit exactly the same behaviour if observed over time.

The compositional strong equivalence aggregation allows a model to be systematically simplified by considering each of its top-level components in turn. Each of these components is treated as a separate model and its top-level components are identified. The process is repeated with the components at the next level and so on. If an identified top-level component is atomic, it is not broken down further into cooperating components. If all top-level components in a model are atomic, the strong equivalence aggregation is applied to the cooperation of these atomic components resulting in a lumped component that replaces them. At each level of the model, the aggregation procedure is applied, until a lumped version of the complete original model is constructed.

4.3 Related Work

Although PEPA has not been as long established as Petri Nets [3, 41, 92, 94], stochastic Petri Nets [4, 19, 24, 25, 77] and queueing theory [28, 69, 70, 80], many researchers have begun to show an interest in using PEPA in systems performance modelling. The following are a few examples that use PEPA to model system performance.

Holton [60] uses PEPA to model and evaluate the performance of an industrial production cell. The production cell has as components a feed belt, elevating rotary table, robot arms, press, deposit belt and traveling crane. The initial model was so large that it was impossible to compute the performance. For this reason, the model has been simplified, reconstructed and evaluated for

its throughput and component utilisation. Various experiments have been carried out on the production cell model by varying the speed of each component in turn. Apart from investigating the effects of the changes on the performance, the experiments also help to detect the bottleneck in the system.

El-Rayes et al [39] investigate the performance of a lift system with the PEPA language. They start with a simple one-person lift operating between two floors, which is served on each floor by a one-person queue. The complexity of the lift system is progressively increased so that the lift can operate between multiple floors serving many-person queues. Various speeds of the lift and the arrival rate of people at each queue have been used to study the mean waiting time of the system. The results obtained from the lift system using PEPA have been compared to those obtained from a traditional engineering method for lift traffic analysis. The authors concluded that the results estimated using PEPA were more consistent and accurate despite some expression restrictions in the PEPA language.

Gilmore et al [46] apply the stochastic process algebra approach to a robot control problem. The task of the robot arm is to receive items repeatedly from a conveyor belt and group them on a pallet, which is removed when it becomes full. The model consists of the conveyor belt, robot arm and the pallet as well as a work-cell plan that is responsible for the communication between the components. Although the performance of the robot control problem has been solved using TIPP [47], the specification of the model actually uses the notations that are available from both TIPP and PEPA. The results produced have been compared to those obtained using other approach such as Petri Nets.

The TIPP (**T**imed **I**nterleaved **P**rocesses **P**erformance Evaluation) [47] is the closest stochastic process algebra language to PEPA. The language also captures some basic interaction patterns of behaviour such as sequential execution, rivalry actions and concurrent execution. These behaviours are represented with *prefix*, *choice* and *parallel composition* combinators. In TIPP, actions are specified as a *(type, rate)* pair. Each action can be either active ((α, λ)) or passive ($(\alpha, 1)$). An active action happens instantaneously after a duration that is exponentially

distributed with rate λ. A passive action, in contrast, waits for a partner before completing the action. A TIPP model is mapped into a Markovian Labeled Transition System (MLTS), which is then transformed into a state transition diagram (infinitesimal generator matrix **Q**) of the underlying continuous time Markov chain. The steady state probability distribution Π can be obtained by solving the linear equation system $\Pi\mathbf{Q} = 0$ subject to $\sum\Pi_i = 1$.

There are two stochastic process algebras called MPA (Markovian Process Algebra). One of these has been developed at the University of Dortmund while the other one has been developed at the University of Bologna. In the MPA of Dortmund [17], actions are represented as (α, r) where α represents the action type. However, it is assumed that every action of type α has a fixed rate of μ_α The r denotes the number of concurrently active instances of the action. There is no passive action in this language.

In the MPA of Bologna [8], actions are represented as *(type, rate)*. Each action can be either active or passive. An active action can be exponentially distributed (α, λ) or instantaneous (α, ∞). A passive action is denoted as $(\alpha, 0)$. When an interaction takes place between two actions, there must be at least one passive action involves. However, interaction between two instantaneous actions is not allowed.

Apart from the notions of equivalence, Hillston [57] has also discussed other relevant techniques that may solve the problem of state space explosion. Amongst those, the product form solution which allows components of a model to be solved in isolation and the partial solution are subsequently combined to give the final solution for the model has been proposed [56, 58]. Nevertheless, the models of this approach concentrated on components of a particular structure that interact in a restricted way to preserve a form of independence between the components.

Chapter 5

GENERAL APPROACH TO MODELLING DBMS WITH PEPA

5.0 Introduction

This chapter and the following few chapters present the modelling of PDBMS and performance evaluation using PEPA. This chapter concentrates on the basic approach to model a system using the PEPA notation via simple examples. Section 5.1 describes and models a simple database system executing on a simple platform. Section 5.2 presents alternative evaluation approaches to estimate the system performance. Section 5.3 extends the system to a more complicated configuration and section 5.4 summarises and concludes the chapter.

5.1 Building A Simple System

To begin with, consider a very simple database system executing on a simple shared-nothing platform which consists of a single processing element (PE) with a single disk attached to it. The system supports a simple query that reads data from a simple database. The simple database has only one relation. In this case, the query scans through the database to search for tuples that match a simple condition.

At this stage, no cache is included in the system. Since the system only performs data reading, system logging is not considered in this model although a simple form of locking is considered. It is also assumed that each transaction will commit once it finishes searching. No transaction abort or transaction rollback is considered.

The processes in the PE consist of a transaction manager, a concurrency control unit, a lock manager and a buffer manager (**Figure 5.1**). The transaction manager (TM) coordinates the transaction requests arriving from the users and passing to the concurrency control unit, and manages the results returning from the concurrency control unit to the users. The concurrency control unit (CCU) ensures the smooth handling of each transaction by scheduling the transaction requests with the help of the lock manager. The lock manager (LM) provides a simple locking system to manage the lock requests from the CCU while the buffer manager (BM) manages the data in the buffer and fetches data from the disk.

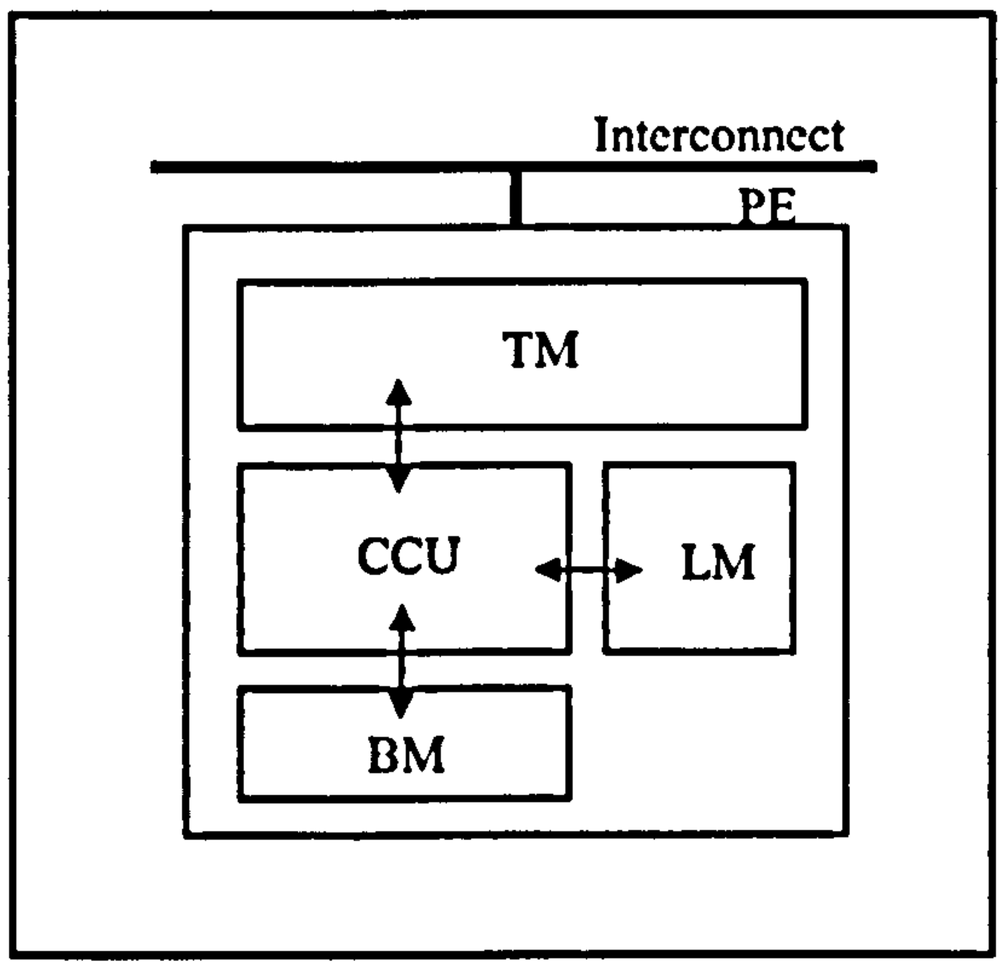


Figure 5.1. Simple DBMS running on a simple platform.

5.1.1 The PEPA Model

One of the main features offered by PEPA is the compositional ability that allows a model to be constructed from its components in a structured way. In this case, the system can be viewed as consisting of four main components (*TM*, *CCU*, *BM*, *LM*) that communicate with each other and the external environment (i.e. the user that will not be explicitly modelled) as shown in **Figure 5.1.**

1. **Component *TM*.** The TM receives transaction requests from users. After receiving a transaction request, the TM sends a message or request to the CCU and waits for the reply from the CCU before relaying the results to the user. In the meanwhile, new transaction requests may arrive and will be dealt with. Thus this component can be written as:

```
#TMa = (request, r_req).(tm2ccu, r_sgn0).TMa;
#TMb = (ccu2tm, infty).(reply, r_reply).TMb;
#TM = TMa <> TMb;
```

The specification of component *TM* shows that the component itself is composed of two components: *TMa* and *TMb*. Component *TMa* represents the situation where the TM receives transaction requests (activity *(request, r_req))* at the rate of *r_req*, and sends messages to the CCU (activity *(tm2ccu, r_sgn0)*) at the rate of *r_sgn0*. The arrival rate of the transaction request is exponentially distributed because it is independent of the time since the last arrival of a transaction request. On the other hand, component *TMb* shows that the TM is waiting for replies from the CCU at an unspecified rate (activity *(ccu2tm, infty)*) because this is a passive activity and it depends on the service rate of the CCU. Then, it relays the results to the users at the rate of *r_reply*. It is assumed that both components *TMa* and *TMb* can proceed concurrently in order to achieve the maximum parallelism within the TM (*TMa* <> *TMb*).

2. **Component *CCU*.** When the CCU receives a request from the TM, it requests the appropriate transaction locks from the LM and waits for the lock(s) to be granted before proceeding with data manipulation. After the lock request is granted, the CCU starts accessing data from the disk via the BM and waits until the BM signals the completion of data transfer. Once this is done, the CCU informs the LM to release the lock(s) held by the completed transaction request and sends a commit message together with the results to the TM.

Since all the transaction requests are read-only, the CCU need not wait until a transaction that is already in process releases its lock(s) and commits, before initiating another transaction request. As soon as the BM completes a request for data transfer and becomes available, the CCU can immediately start another such request.

While the CCU is waiting for the lock request to be granted, or waiting for the BM to complete the data transfer, new requests may arrive from the TM. Thus, component CCU can be expressed as:

#Qa = (tm2ccu, infty).(ccu2lm, r_sgn0).Qa;
#Qb = (lm2ccu, infty).(begin_trans, r_sgn).Qb;
#Q = Qa <> Qb;
#P = (begin_trans, infty).(ccu2bm, r_sgn0).P;
#R = (bm2ccu, infty).(ccu2lm0, r_sgn0).(ccu2tm, r_commit). R;
#CCU = Q < begin_trans> P <> R;

There is an internal cooperation that takes place between component *Qb* and component *P*. When the CCU receives a lock granted message from the LM, it will start a new data transfer via the BM so long as the BM is free. If the BM is engaged with another data transfer activity at this time, the CCU has to wait until the BM becomes available before it can start another data transfer. When this happens, the CCU will not be ready to receive any lock granted message from the LM. Consequently, the LM will be unable to send any lock granted message to the CCU until the CCU is ready. For this reason, component *Qb* is introduced to handle the lock granted messages from the LM while component *P* can concentrate on feeding the transaction requests to the BM in order to minimise the overall waiting time between component LM and component CCU.

3. **Component *LM*.** When the LM receives a request from the CCU, it locks the page, grants the lock request, and returns a message to the CCU. Since all the requests are read only, they can share the page lock. Once a lock request has been granted and a message has been sent, the LM waits for the lock release message from the CCU in order to release the lock after the transaction has completed. Any new lock requests that arrive in the interim are dealt with in the same way. The PEPA specification of this component is straightforward and can be represented as:

#LMa = (ccu2lm, infty).(lm2ccu, r_gnt).LMa;
#LMb = (ccu2lm0, infty).(release, r_sgn).LMb;
#LM = LMa <> LMb;

4. **Component BM.** The BM waits for requests from the CCU. When the BM receives a request from the CCU, it starts reading data from the disk and delivers the result. It is assumed that for every transaction request, the BM has to fetch all pages from the disk.

$$\#BM = (ccu2bm, infty).(deliver, r\_deliver).(bm2ccu, r\_sgn0).BM$$

The main components of the system have been constructed separately. This allows one to concentrate on building the specification of a component without worrying about the details of other components. These components can now be linked together to form a model for the system. Although these components have to interact amongst each other to complete a transaction request, they also perform some activities independently when appropriate. Hence, the model can be expressed as follows;

$$\#Model = (TM <> BM <> LM) \\ <tm2ccu, ccu2tm, bm2ccu, ccu2bm, ccu2lm, ccu2lm0, lm2ccu> CCU;$$

5.1.2 Evaluating the System Performance

To evaluate the performance of the system, the underlying stochastic process has to be generated. This can be achieved with the PEPA Workbench. The workbench transforms the PEPA specification into the underlying stochastic process which is represented by an infinitesimal generator matrix **Q**. The model is processed and has 1156 states.

The system performance throughout this study is measured in terms of throughput expressed as transaction per second (tps). In terms of PEPA, the throughput for this model is the product of the probability that activity *(reply, r_reply)* is enabled and the activity rate, *r_reply*.

$$\text{throughput } Model = \sum \Pi(X_i).r\_reply; \\ X_i \in \{\text{states where activity } (reply, r\_reply) \text{ is enabled}\}$$

Several experiments have been conducted by varying the values of the parameters or variables of the matrix **Q** and the formula.

5.1.3 Experiments

The database consists of a single relation (*Account*) with a tuple size of 100 bytes. The page size is set to 1024 bytes and each page is assumed to be 70% full (or 716 bytes each). Thus, each account relation page will have 7 tuples (700 bytes). The relation is broken into a number of fragments and for simplicity, each fragment is assumed to consist of a single page. The average time to read a page from the disk is taken to be 41.11 milliseconds (based on the figures in [111]).

For this experiment, the database size is varied by changing the number of relation fragments (pages) allocated to the disk. Because the PEPA model does not explicitly model the database size, the number of relation fragments is reflected in the data delivery rate of the BM (i.e. activity *(deliver, r_deliver)*). To deliver a page, it takes 41.11 milliseconds or 24.325 pages per second. The arrival rate of transaction requests is set to 100, 25, 10 and 5. **Table 5.1** summarises the results obtained by solving the PEPA model for these values.

Query arrival rate:	No. of Fragments on the disk			
	1	2	3	4
100	24.203	12.133	8.095	6.074
25	21.058	12.051	8.089	6.073
10	9.961	9.252	7.532	5.950
5	4.997	4.983	4.888	4.629

Table 5.1. System throughput (tps) for the simple database system.

The results show that when the query arrival rate is 100 and the relation consists of a single page stored on disk, the estimated system throughput is 24.203 tps. For the same arrival rate, the estimated system performance is 12.133 tps when the relation consists of two pages, and so on.

Different query arrival rates also affect the overall average system performance. For instance, when the query arrival rate is much greater than the maximum throughput of the system, the average system throughput is about the same as the maximum throughput. This is because the arriving of requests is faster than the processing speed of the system and thus causes the requests to wait in the queue. When the query arrival rate is about the same as the maximum throughput, the average system throughput is slightly lesser than the maximum system throughput. This is because the requests arrive randomly. Sometimes the requests may pack the system until some of them are blocked waiting in the queue. While other times, the system is so idle that the arriving request can be processed immediately. When the query arrival rate is much lower than the maximum throughput, the average system throughput is the same as the query arrival rate as the requests can be processed as soon as it arrives.

5.2 Model Simplification and Model Decomposition

Clearly the model in the previous section is far too simplistic and we need to move to more realistic models. However, as the model becomes more complicated, for instance by introducing a cache to the buffer manager; or performing an update operation; or adding a new component (e.g. PE), the size of the state space underlying the model grows rapidly. This problem is known as state-space explosion and it may result in a set of states that is too large to be solved. To overcome this problem, various notions of equivalence are introduced in [53]. Hereafter, the model in the previous section will be referred to as *ModelA*.

5.2.1 Using Compositional Strong Equivalence Aggregation Approach to Reduce State-Space

ModelA has four components at its top-level: *TM*, *CCU*, *BM* and *LM*. Component *TM* is composed of two atomic components (*TMa* and *TMb*), whereas component *CCU* consists of two

atomic components (*P* and *R*) and one composed component (i.e. component *Q* that consists of atomic components *Qa* and *Qb*). Component *LM* is composed of components *LMa* and *LMb* while component *BM* is itself an atomic component.

The strong equivalence aggregation can be applied to the cooperation of the atomic components *TMa* and *TMb* and results in a lumped component to replace them. As a result, component *TM* can be expressed as:

#TM = (request,r_req).TM1 + (ccu2tm,infty).TM2;
#TM1 = (tm2ccu,r_sgn0).TM + (ccu2tm, infty).TM3;
#TM2 = (request,r_req).TM3 + (reply,r_reply).TM;
#TM3 = (tm2ccu,r_sgn0).TM2 + (reply,r_reply).TM1;

The same procedure is applied to the cooperation of the atomic components *Qa* and *Qb* and a lumped model of *Q* is formed.

#Q = (tm2ccu, infty).Q1 + (dlm2ccu,infty).Q2 ;
#Q1 = (ccu2dlm, r_sgn0).Q + (dlm2ccu,infty).Q3;
#Q2 = (tm2ccu, infty).Q3 + (begin_tran , r_sgn).Q;
#Q3 = (ccu2dlm, r_sgn0).Q2 + (begin_trans, r_sgn).Q1;

Lumped components *CCU* and *LM* are formed in the same fashion and at the final stage, the strong equivalence aggregation procedure is applied to the cooperation of the top-level components of *ModelA* (lumped components *TM*, *CCU* and *LM*, and atomic component *BM*). A lumped version of *ModelA* is formed **(Appendix A)**.

The lumped *ModelA* is processed with the PEPA workbench and its underlying stochastic process is generated and is captured in a matrix **Q**. The lumped model has 80 states and has been significantly reduced compared to the original *ModelA* (1156 states). The throughput of the lumped model is measured in the same way when estimating the throughput of *ModelA*. The same set of experiments conducted on *ModelA* is carried out for lumped *ModelA*. The results are summarised in **Table 5.2**.

Query arrival rate:	No. of Fragments on the disk			
	1	2	3	4
100	24.321	12.163	8.108	6.081
25	21.129	12.080	8.102	6.080
10	9.972	9.264	7.542	5.957
5	4.999	4.986	4.891	4.632

Table 5.2. System throughput (tps) for lumped *ModelA*.

The results obtained using lumped *ModelA* are fairly similar to those estimated by solving the original *ModelA* (with average differences lesser than 1%). Nevertheless, the lumped version of *ModelA* does not reflect the actual behaviour of the system as clearly as the original model.

Although the compositional strong equivalence aggregation can reduce the state-space of *ModelA* significantly, when the system is extended and its complexity grows it may be difficult to follow the aggregation procedure to form a complete lumped model. In particular, the state-space may be too large to be reduced sufficiently to be able to obtain a solution for the model.

An approach is needed that can be applied to model a simple database system which has only one PE, as well as to model a parallel database system with multiple PEs. In addition, this approach should allow the model to reflect the behaviour of the system as clearly as possible.

5.2.2 Decomposing *ModelA*

The idea of the decompositional approach is inspired by the *flow-equivalent aggregation* technique for a queueing network [67, 69, 96], which is based on Norton's theorem for electrical circuit analysis [21, 22]. According to this technique, the job flow through an aggregated queue/composite queue is equivalent to the job flow through the respective subnetwork (**Figure 5.2**). This approach can also be applied to an open network [21] (**Figure 5.3**). In this case, the job flow into the composite queue (job arrival rate) is equivalent to the output rate (throughput) of the server.

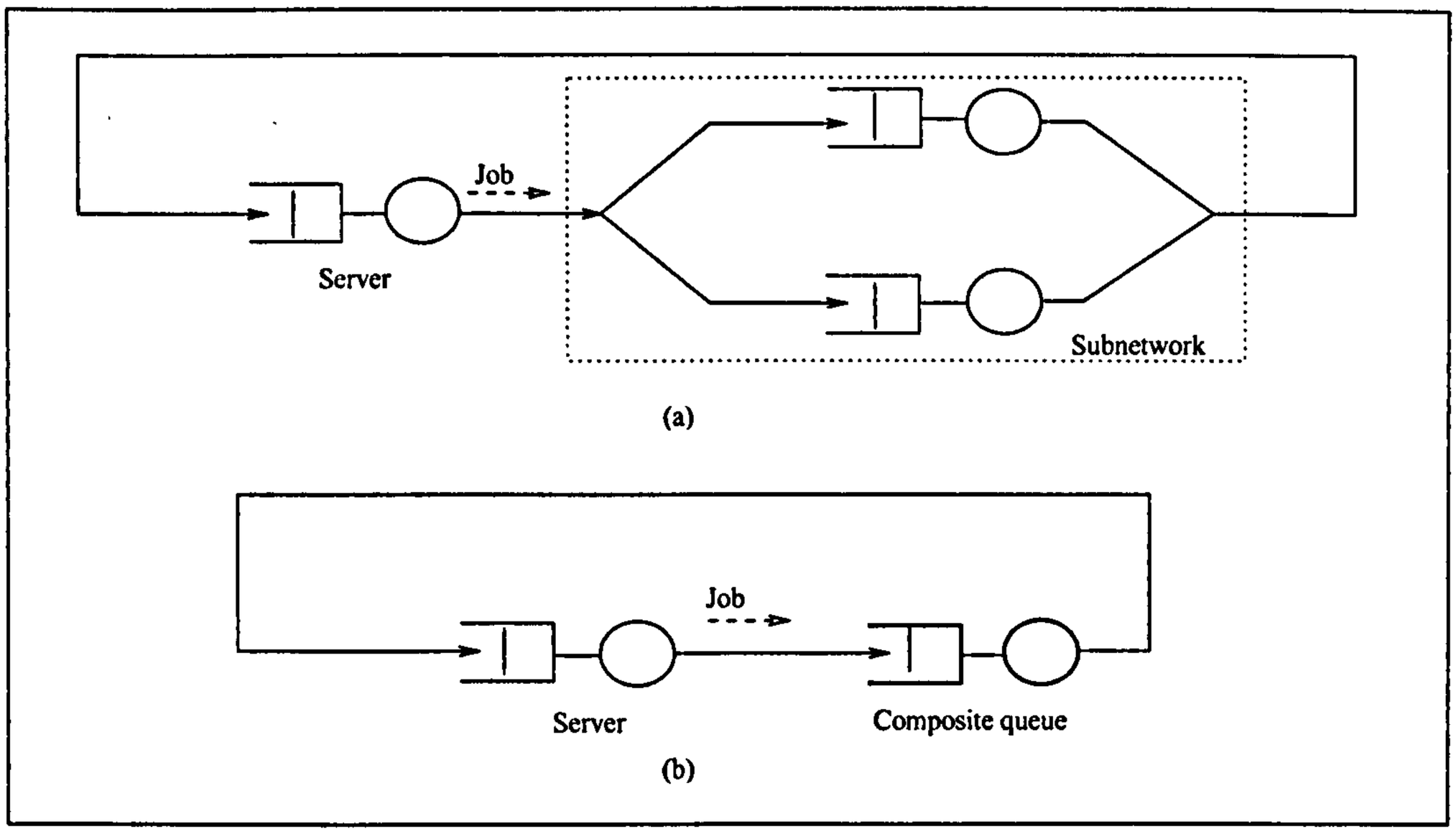


Figure 5.2. a) Original network. b) Aggregated network.

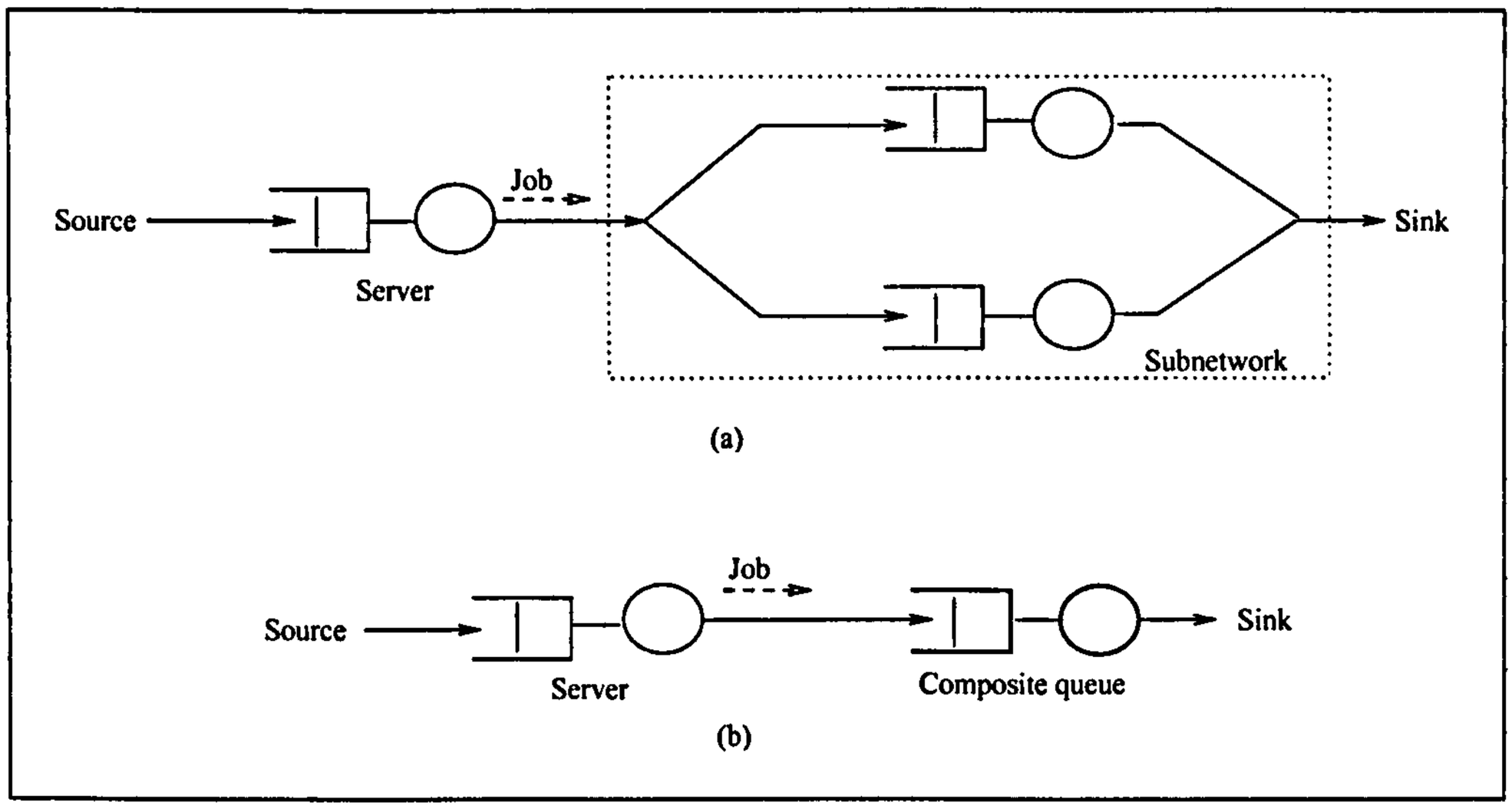


Figure 5.3. a) Original open network. b) Aggregated open network.

Consider a PEPA model as below (**Figure 5.4**). The request flow from component *a* to component *b* in **Figure 5.4(a)** is equivalent to the request flow from component *a* to the aggregated component *x* in **Figure 5.4(b)**. This exhibits the concept of flow rate and is analogous to the theorem of the flow equivalent aggregation technique. Component *a* therefore can be isolated from the model and evaluated like a PEPA model. The throughput of this submodel will become the request arrival rate of component *b*. The same idea can also be applied to component

b, *c* and *d* in turn, which can be broken into submodels that are subsequently evaluated in isolation. The throughput of each submodel becomes the request arrival rate of the subsequent submodel.

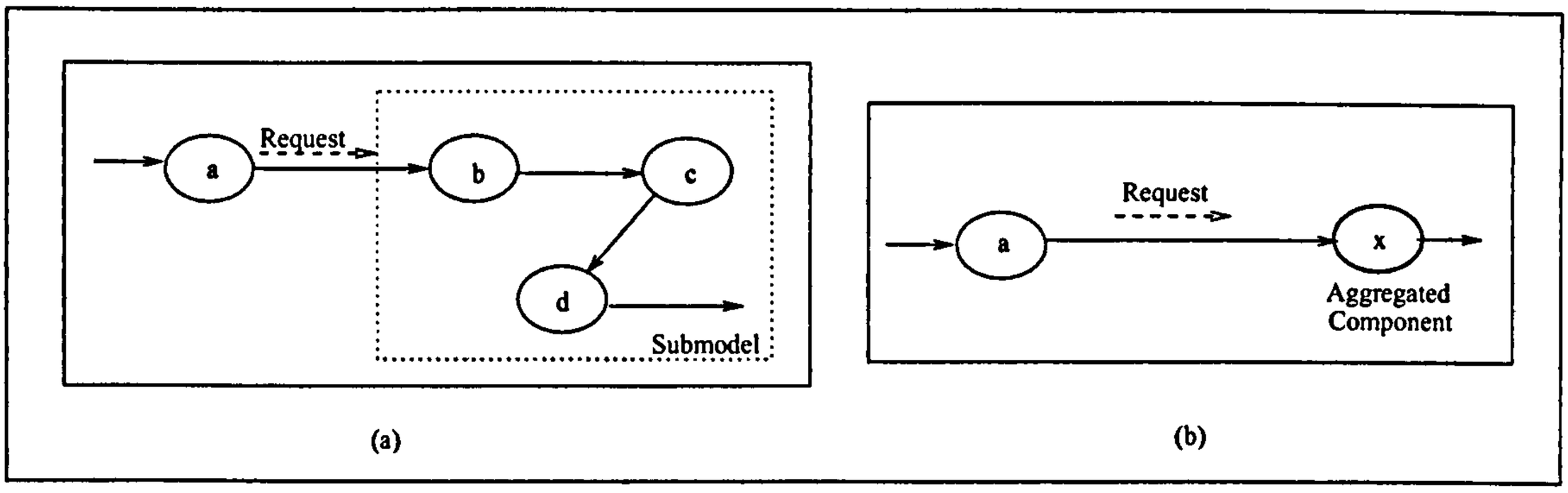


Figure 5.4. a) Original PEPA model. b) Aggregated model.

For example, *ModelA* can be decomposed into eight submodels (*TMa*, *TMb*, *CCUa*, *CCUb*, *CCUc*, *LMa*, *LMb* and *BM*) each consisting of an atomic component, except submodel *CCUb* **(Figure 5.5 (a) and (b))**. Submodel *CCUb* has not been further decomposed into submodels (e.g. submodel *Qb* and submodel *P*) because the activity between component *Qb* and component *P* (*(begin_trans, r_sgn)*) is an internal activity which is not observable to other submodels (e.g. submodel *LMa* and submodel *BM*).

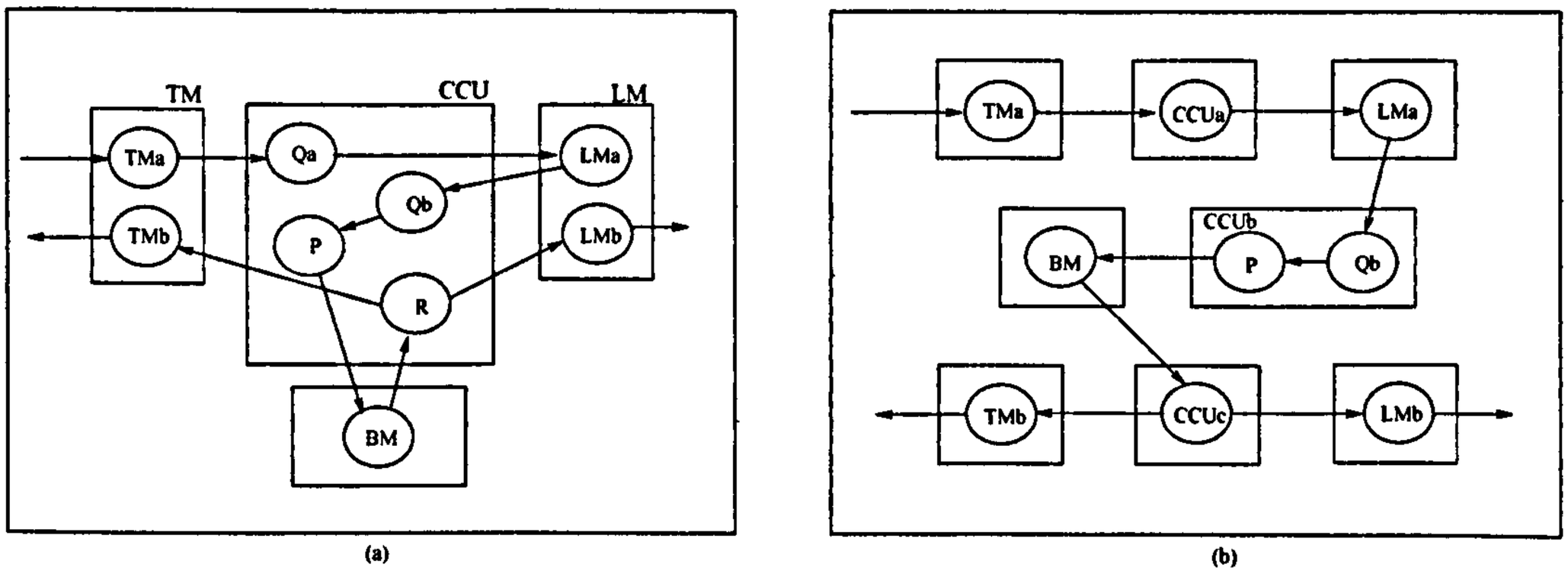


Figure 5.5. (a) *ModelA*, (b) Decomposed *ModelA*.

These submodels are evaluated separately in turn starting with submodel *TMa*. The throughput of submodel *TMa* becomes the activity rate r_i in activity *(tm2ccu, r_i)* of submodel *CCUa*, and the throughput of submodel *CCUa* subsequently becomes the activity rate r_j in activity *(ccu2lm, r_j)* of submodel *LMa*, and so on until all submodels are evaluated (**Figure 5.5(b)**). This process is repeated until all submodels are evaluated.

However, a problem was noticed with this approach in that the overall system throughput estimated was actually different from that achieved with the complete model. For example, for the case of 1 relation page with a transaction request arrival rate of 100, the result produced is approximately 5% lesser than the performance obtained by solving the original model in full (23.130 tps compared to 24.203 tps). This was attributed to the fact that before *ModelA* is decomposed into submodels, there is an underlying pipeline parallelism effect among the components. However, there is little or no such effect in the submodels of the decomposed *ModelA*. For this reason, a *queue generator* which simulates the arrival of requests from a component which requests the service of the current component has been introduced to every submodel except the one that interacts with the external environment (i.e. submodel *TMa* that receives requests from the users). As an example, in the case of *ModelA*, component *TMa* of the TM sends requests to the CCU (via component *Qa*) for further processing. When these components are broken into two submodels, a queue generator is placed in submodel *CCUa* (which consists of component *Qa*) in order to simulate the flow of requests from component *TMa*. The throughput of submodel *TMa* becomes the arrival rate for the queue generator. The specification of submodel *CCUa* with a queue generator (represented by $\#Q_i$) is as follow.

$\#Q_0 = (arrival, lambda).Q_1;$
$\#Q_n = (arrival, lambda).Q_{n+1} + (tm2ccu, infty).Q_{n-1};$
$\#Q_N = (tm2ccu, infty).Q_{N-1};$

$\#Qa = (tm2ccu, rs).(ccu2lm, r\_sgn).Qa;$
$\#CCUa = Qa \langle tm2ccu \rangle Q_0;$

where $N \in \mathbf{I}^+;$ $n \in \{1..N-1\};$

Each of the submodels except submodel *TMa* is now attached to a queue generator **(Appendix B)**. The submodels are re-evaluated in turn starting with submodel *TMa*. The process is iterated until it converges on a solution. The resulting throughput for submodel *TMb* is obtained and this throughput also represents the overall system throughput of the model. The results collected this time **(Table 5.3)** compare favourably with those obtained from solving the original *ModelA*. Hereafter, the adapted decompositional approach including queue generators is referred as the decompositional evaluation approach.

Query arrival rate:	No. of Fragments on the disk			
	1	2	3	4
100	24.204	12.133	8.095	6.074
25	21.061	12.051	8.089	6.073
10	9.961	9.253	7.532	5.950
5	4.998	4.984	4.889	4.629

Table 5.3. System throughput (tps) for decomposed *ModelA*.

5.2.3 Comparing Strong Equivalence Aggregation and Decompositional Evaluation Approach

Figure 5.6 displays the system performance of *ModelA* obtained by using three different approaches. The graphs show that from this case there is little difference between the results estimated using compositional strong equivalence aggregation technique and those obtained from the decompositional evaluation approach, compared to the results estimated by solving the full set of states of *ModelA*(refer **Table 5.1, 5.2** and **5.3** for figures). At this state, it may be safe to say that when the state-space is too large to obtain a solution, either the compositional strong equivalence aggregation technique or the decompositional evaluation approach can be used.

Although both the compositional strong equivalence aggregation technique and the decompositional evaluation approach can reduce the state-space of *ModelA* to a manageable size, the latter method has a slight advantage over the former one. The decomposed submodels clearly

reflect the behaviour of the system as individual components, as well as the communication between the components while the lumped model may be more difficult to understand. Furthermore, transforming the system components into decomposed components is fairly straightforward compared with transforming the system components into lumped components and later to a lumped version of the model.

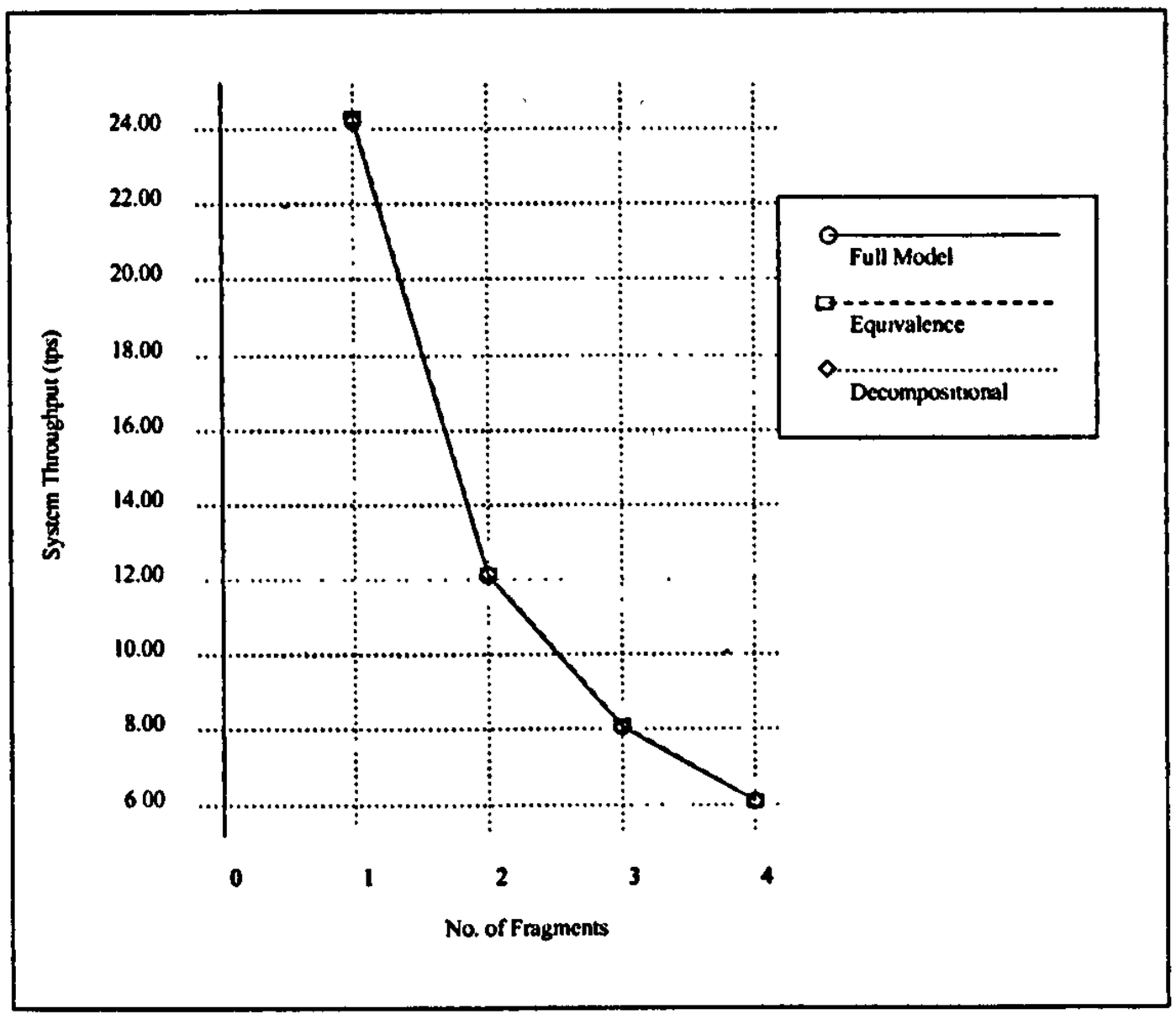


Figure 5.6. System throughputs (tps) for *ModelA* obtained using various approaches for the case when the arrival rate of requests is 100.

5.3 Extending the Simple Model

The previous section has demonstrated how PEPA is used to model a simple database system and how the system performance is evaluated. The system is now extended to handle two disks on a single PE. Apart from the number of disks attached to the PE, the structure of the system remains the same as the one in the previous section.

5.3.1 Modelling Single Node with Multiple Disks

The modelling process for this system is quite straightforward because most of the descriptions of the components of this model are similar to those in *ModelA*. Therefore, it is possible to reuse most of the PEPA specification of *ModelA* for this model. The only modification that has to be made is confined to component *CCU* and component *BM*.

When the CCU receives a lock-granted message from the LM, it signals the BM to read data from both disks, and waits for the BM to return the signals after accessing the data from the disks. After that, the CCU sends messages to the LM and TM respectively to release the locks and commit the transaction. The BM has two components, each handling one disk. It is assumed that when the BM receives the signal from the CCU, it reads data from the two disks simultaneously (i.e. the two components that handle the disks will proceed in parallel). The BM sends a message to the CCU after reading the data from both disks. Hence, component *CCU* and component *BM* of the new model, *ModelB*, can be expressed as follows:

```
#Qa = (tm2ccu, infty).(ccu2lm, r_sgn0).Qa;
#Qb = (lm2ccu, infty).(begin_trans, r_sgn).Qb;
#Q =  Qa <> Qb;
#P = (begin_trans, infty).P';
#P' = (ccu2bmA, r_sgn0).(ccu2bmB, r_sgn0).P +
        (ccu2bmB, r_sgn0).(ccu2bmA, r_sgn0).P;
#R = (bmA2ccu, infty).(bmB2ccu, infty).R' +
        (bmB2ccu, infty).(bmA2ccu, infty).R';
#R' = (ccu2lm0, r_sgn0).(ccu2tm, r_sgn0).R;
#CCU = Q <begin_trans> P <> R ;

#BMa = (ccu2bmA, infty).(deliver, r_deliver).(bmA2ccu, r_sgn0).BMa;
#BMb = (ccu2bmB, infty).(deliver, r_deliver1).(bmB2ccu, r_sgn0).BMb;
#BM = BMa <deliver> BMb;
```

ModelB is processed (with the PEPA workbench) and has 1952 states. The same set of experiments carried out on the previous model is conducted on this model. Since there are two disks attached to a single PE, the database has to be equally distributed among the disks. This is achieved by allocating the relation fragments to the disks in a round-robin fashion. **Table 5.4** summarises the results of the experiments.

Query arrival rate:	No. of Fragments on the disks						
	$Disk_0$	1	2	2	3	3	4
	$Disk_1$	1	1	2	2	3	3
100		24.160	12.122	12.122	8.090	8.090	6.071
25		21.042	12.041	12.041	8.084	8.084	6.070
10		9.961	9.250	9.250	7.529	7.529	5.948
5		4.997	4.983	4.983	4.888	4.888	4.628

Table 5.4. System throughput (tps) for *ModelB*.

Lumped *ModelB* and Decomposed *ModelB*

The lumped version of *ModelB* is formed following the compositional strong equivalence aggregation procedures. However, component *BM*, which consists of components *BMa* and *BMb*, is left unlumped because the lumped version of component *BM* did not reflect the synchronisation of both components *BMa* and *BMb*. Consequently, the data delivery activity would not be evaluated when evaluating the overall system performance for the lumped *ModelB*. Nevertheless, the selective lumped *ModelB* is still strongly equivalent to the original model [53] **(Appendix C)**.

The decomposed version of *ModelB*, like the original *ModelB*, recycles most of the PEPA specifications used in the decomposed *ModelA*, except submodels *CCUb*, *CCUc* and *BM*. Nevertheless, submodels *CCUb* and *BM* can be easily transformed from components *CCUb* and *BM* of *ModelB*. As for submodel *CCUc*, it has to be able to capture the synchronisation of the disk units, as well as the flow of messages from the BM. For this reason, it is expressed as follows:

$$\#Q_0 = (arrival, lambda_0).Q_1;$$
$$\#Q_n = (arrival, lambda_0).Q_{n+1} + (bmA2ccu, infty).Q_{n-1};$$
$$\#Q_N = (bmA2ccu, infty).Q_{N-1};$$

$$\#K_0 = (arrival, lambda_1).K_1;$$
$$\#K_n = (arrival, lambda_1).K_{n+1} + (bmB2ccu, infty).K_{n-1};$$
$$\#K_N = (bmB2ccu, infty).K_{N-1};$$

#P = (bmA2ccu, r_sgn).(bmB2ccu, r_sgn).P' +
(bmB2ccu, r_sgn).(bmA2ccu, r_sgn).P';
#P' = (ccu2lm0, r_sgn0).(ccu2tm, r_commit).P;
#CCUc = (Q_0 <arrival> K_0) <bmA2ccu, bmB2ccu> P;

where $N \in \mathbf{I}^+$; $n \in \{1..N-1\}$;

Note that in submodel *CCUc*, the synchronisation of the disk units is reflected in the cooperation between the two queue generators Q_0 and K_0 in activity *(arrival, lambda$_i$)*. The PEPA specification of decomposed *ModelB* is attached in **Appendix D.**

Both lumped version and decomposed version of ModelB are processed. The lumped *ModelB* has 318 states while the submodel *BM* of decomposed *ModelB* (which has the biggest state space among the submodels, when $N = 5$) has 50 states. Despite the different approaches, the results obtained from the two methods agree closely with those obtained from solving the full *ModelB* using the PEPA approach **(Figure 5.7).**

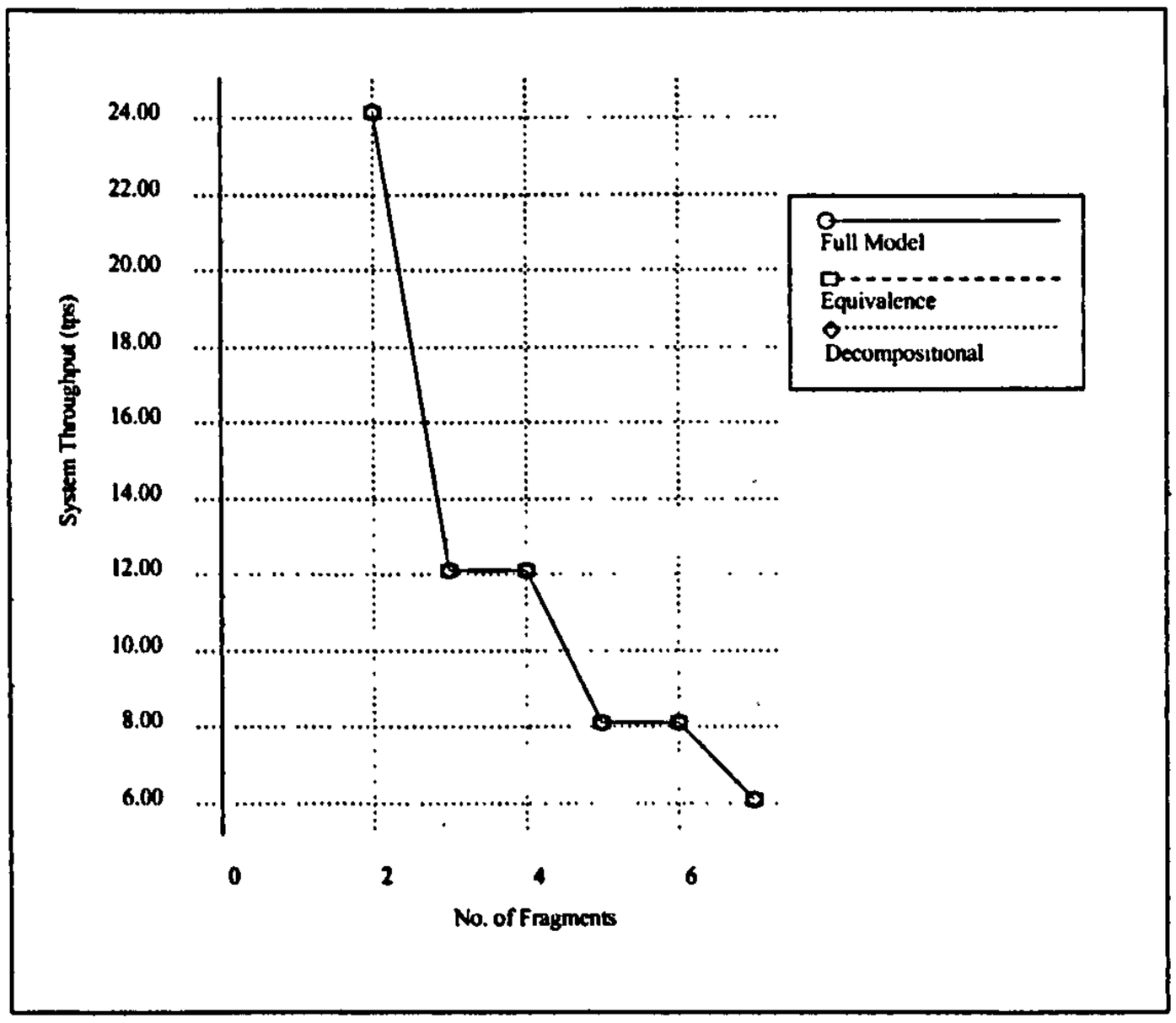


Figure 5.7. System throughput (tps) for *ModelB* obtained using various approaches for the case when the arrival rate of requests is 100.

5.3.2 Modelling Double Nodes with Single Disk Each

The complexity of the system is further increased by increasing the number of PEs to two, with a single disk attached to each. The system that runs on each PE still consists of one transaction manager, one concurrency control unit, one distributed manager and one buffer manager. The responsibility of each component remains the same except for the transaction managers.

When the transaction manager (TM_0) of PE_0 receives a transaction request from a user, it sends one request to the local concurrency control unit (CCU_0) and another to the remote transaction manager (TM_1) on PE_1. TM_0 waits for the replies from both CCU_0 and TM_1 before replying to the user. In the meanwhile, new transaction requests may arrive. It is assumed that TM_0 acts as a transaction host that receives requests from users and subsequently distributes such requests among the PEs. The replies are also handled in this way.

The transaction manager (TM_1) of PE_1 does not receive requests directly from the user but receives them from the TM_0. It forwards each request to its concurrency control unit (CCU_1) on PE_1 and waits for the commit message. It sends a reply to TM_0 once it receives the message from CCU_1. Meanwhile, TM_1 may receive further requests from TM_0.

Transforming the system into a PEPA model is quite straightforward. This can be done by duplicating the PEPA model of *ModelA* (which represents a system with a single PE) to two *ModelA*s, and put them together after some modification to components TM_i. The new model represents a system that has two PEs, each has a single disk attached to it, and is referred to here as *ModelC* **(Appendix E).**

However, the state-space of *ModelC* is too large (approximately 29 million states) to be resolved by any mathematical tool. For this reason, either the compositional strong equivalence aggregation method or the decompositional evaluation approach must be used to estimate the system performance. For the same reasons, the lumped version of *ModelC* is too complicated to be constructed and its specification has also become less meaningful (approximately 100000

states). As a result, the system performance evaluation has been computed using the decompositional evaluation approach.

Each PE is decomposed as for *ModelA* although there are differences in components TM_i. Each submodel of the decomposed *ModelC* is evaluated in turn starting from submodel TM_0a. The throughput of submodel TM_0a becomes the arrival rate of requests for submodels CCU_0a and TM_1a. Apart from submodels TM_i of both PEs, the remaining submodels of both PEs have no interaction across the PEs and thus can proceed in parallel. **Table 5.5** summarises the experimental results. This is to confirm that the results obtained using the decompositional evaluation approach are in good agreement to those obtained by solving the PEPA model in full.

Query arrival rate:	No. of Fragments on the disks						
	PE_0	1	2	2	3	3	4
	PE_1	1	1	2	2	3	3
100		22.184	12.130	11.121	8.063	7.420	6.008
25		19.282	12.039	11.045	8.055	7.414	6.006
10		9.122	8.748	8.475	7.370	6.902	5.850
5		4.578	4.572	4.566	4.520	4.479	4.346

Table 5.5. System throughput (tps) for *ModelC*.

5.4 Decompositional Evaluation Approach

In the previous section, several PEPA models have been evaluated following the decompositional evaluation approach. This section summarises the heuristic we follow when evaluating the PEPA models using this approach.

1. Given a full PEPA model, identify the request flow between each component. Isolate these components into submodels each consists of preferably an atomic component.

2. For each submodel, which corresponds to a component that receives requests from another component (but not from the users), include a queue generator that captures the arrival of such requests to the submodel. If a component receives requests from more than one component, represent each of these request flows with one queue generator. Modify each submodel if necessary in order to reflect the behaviour of the component as it is in the original model. The size of the queue generator depends on the number of components a request flows through before arriving at the current component. A length of five elements is sufficient to reflect this, however. Also note that if the size of the queue generator is too long, the size of the state space underlying the submodel can grow rapidly.

3. When the submodels are ready, they are evaluated in turn starting with the submodel that corresponds to the component that receives requests from the users. Each of these submodels will be evaluated like an ordinary PEPA model. The throughput of each submodel becomes the request arrival rate at a queue generator of a subsequent submodel. This process is repeated for every submodel in turn until the final submodel (which corresponds to a component that returns results to the users) is evaluated. The throughput of this submodel becomes the throughput of the overall system performance. The order of evaluating the submodels follows the sequence of the request flowing through the components in the model. However, if there are components performing their tasks in parallel, submodels that are corresponding to these components can be evaluated in any order on their turn.

4. There are situations when this approach may not converge on a solution. This can happen when there is a cyclic/recursive dependency amongst several submodels (which correspond to components that receive feedback from other components). For instance as shown in **Figure 5.8(a)**, submodel *b* depends on submodel *a* for the requests arrival rate while submodel *c* depends on the throughput of submodel *b* before it can be evaluated. However, submodel *a* also depends on the feedback rate from submodel *c* besides the requests arrival rate from other element(s). In this situation, the throughput of submodel *a* will not be reliable until the feedback rate of submodel *c* is known. However, submodel *c* will not generate a

proper throughput until the throughput of submodel *b* is computed, of which in turn depends on the throughput of submodel *a*. In this case, the only solution is to group the components into one submodel (**Figure 5.8(b)**). If component *a* used to receive requests from other component(s), the queue generator(s) will be attached to this lumped submodel. If component *a* receives requests directly from the user, no queue generator is needed.

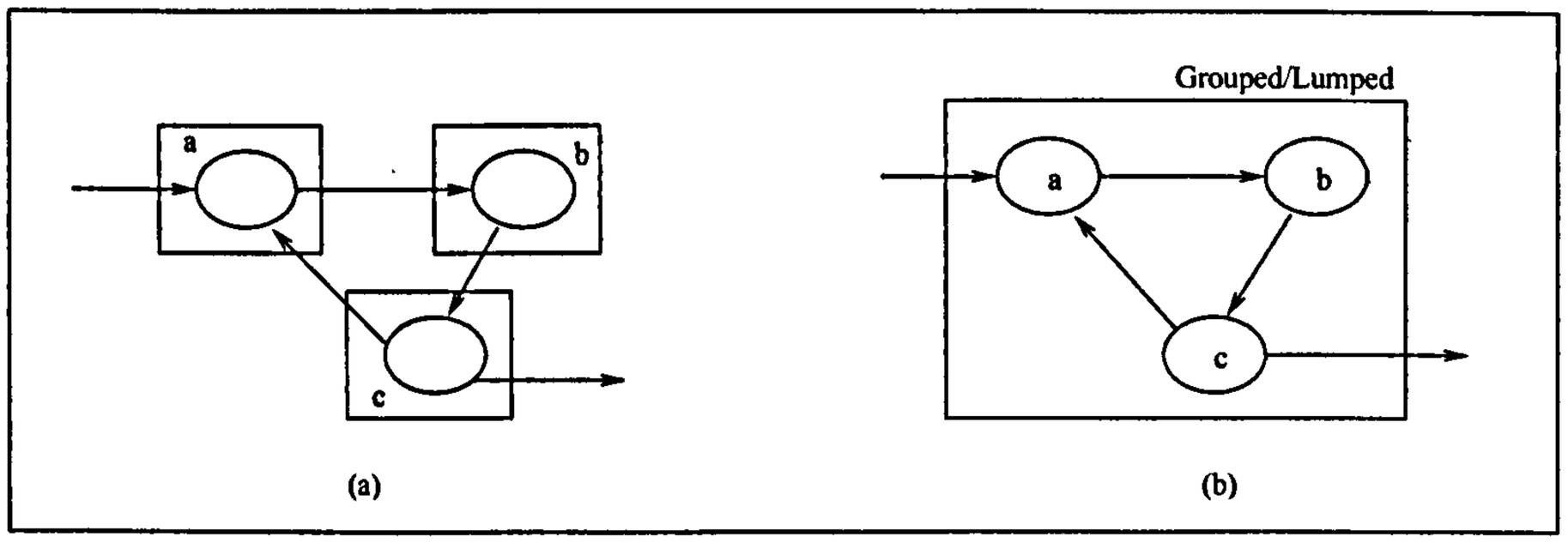


Figure 5.8. a) Submodels *a*, *b* and *c*. b) Lumped submodel of components *a*, *b* and *c*.

5. Sometimes, the complexity and the size of a submodel (which consists of an atomic component) can be very large and the size of the state space underlying the submodel is too large to converge on a solution. If this is the case, the submodel has to be further decomposed into smaller models. This can be achieved by applying the intermediate threading technique that takes advantage of the underlying parallelism amongst the activities of the submodel, if any. This technique will be discussed in length via examples in *Chapter 6*. If this problem occurs in a lumped submodel that consists of several atomic components (because of the cyclic dependency), the compositional strong equivalence aggregation technique can be applied to these components in order to reduce the state space underlying the submodel. However if both of these techniques cannot resolve the problem, the other option will be to reconfigure the system modelled.

5.5 Summary

This chapter has demonstrated how PEPA can be used to model and evaluate the performance of a simple database system. It also exploits the compositional capability and the flexibility offered by PEPA when constructing each of the three models: *ModelA*, *ModelB* and *ModelC*. The only drawback so far is the state-space explosion that occurs as the system becomes more complicated. As a result, the compositional strong equivalence aggregation technique and the decompositional evaluation approach have been used to assist in obtaining solutions.

The examples and experiments conducted in this chapter have shown that the decompositional evaluation approach has slight advantages over the strong equivalence aggregation in terms of simple and meaningful notations. More importantly, it produces results that compare favourably to those obtained from solving the original model in the full PEPA approach. Nevertheless, the compositional strong equivalence aggregation method can be incorporated into the decompositional evaluation approach when appropriate (that is when the size of a submodel that consists of several atomic components is too large to converge on a solution).

In the following chapters where PEPA will be used to model more complicated systems, only the decompositional evaluation approach will be used to evaluate the system performance although the models are constructed following the PEPA approach.

Chapter 6

MULTIPLE NODE DBMS MODELLING WITH PEPA

6.0 Introduction

The previous chapter discussed the basic approach that was followed when using PEPA to model a database system. This chapter takes a step further to introduce the concept and application of intermediate threading for modelling parallel database systems with multiple nodes using the PEPA language. The following section studies a system with four processing elements each with a single disk attached to it. Section 6.2 extends the study to models that have 8 PEs and 16 PEs respectively. Section 6.3 summarises the discussion.

6.1 Multiple Nodes with Single Disk Each

The database system studied in the previous chapter (*ModelA*) supports a simple query that reads data from a database. This section considers running this on a shared-nothing parallel platform that has four PEs, each with a single disk attached to it. In this case, the database is distributed across the PEs following a round-robin data distribution strategy. The query requires the system to read data from all PEs before committing a transaction.

The structure of the system running on each PE still consists of one transaction manager, one concurrency control unit, one lock manager and one buffer manager (**Figure 6.1**). However, there are some changes in the role of the transaction managers.

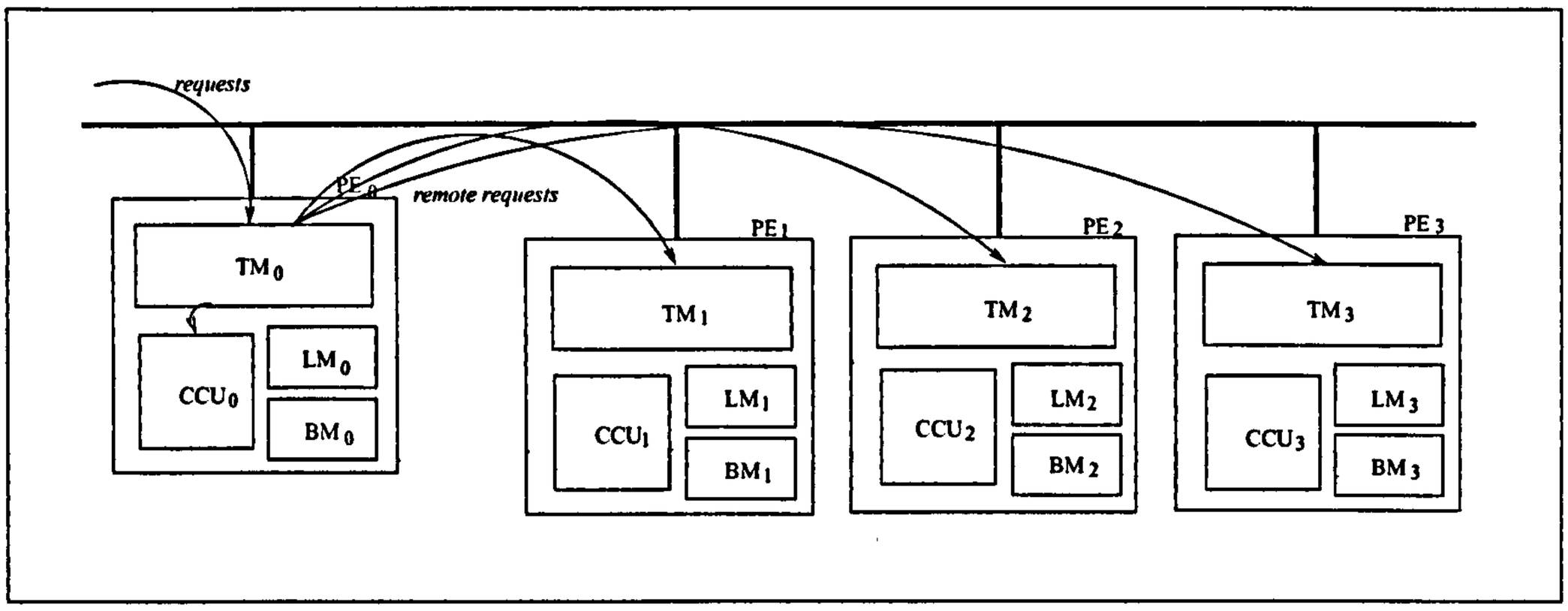


Figure 6.1. System with 4 processing elements.

To make things simple, it is assumed that all transaction requests are handled by the transaction manager on PE_0 (TM_0). When TM_0 receives a request from a user, it sends one request to the local concurrency control unit (CCU_0), and one request each to the transaction managers on PE_1, PE_2 and PE_3 respectively. TM_0 waits for the reply from CCU_0 and those from TM_1, TM_2 and TM_3 respectively. TM_0 has to ensure that it receives all four replies from these components before committing a transaction and replying to the user. In the meantime, new transaction requests may arrive.

The transaction managers on PE_1, PE_2 and PE_3 wait for the requests from TM_0. When a transaction manager receives such a request, it forwards this request to its respective concurrency control unit and waits for the reply from this component before sending messages to the TM_0.

The functions or behaviour of the CCU_i, LM_i and BM_i are similar to those in *ModelA*. When the CCU_i receives a request from the TM_i, it sends a lock request to the LM_i and processes the data via the BM_i after the lock request has been granted. And so on.

Note that the PEs may proceed in parallel when they do not synchronise in any process besides the interaction between transaction managers.

6.1.1 The PEPA Model

The PEPA model for this system can be constructed in a structured fashion starting from the components of each PE. Since the PEs are homogeneous and share similar behaviour, it is sufficient to construct only one model for a PE and then use it to evaluate the performance of each PE in turn although the responsibility of TM_0 and TM_i are different. In addition, the specification of the PE model is similar to the specification of PE_1 of *ModelC* in the previous chapter and thus it can be used in this case.

As for TM_0, the specification should be able to establish the behaviour of the component when it sends requests to CCU_0 and the TM_i as well as when it receives the replies from these components. Hence, TM_0 can be expressed as follows:

$$\#TM_0a = (request, r\_req).TM'_0a;$$
$$\#TM'_0a = (tm2ccu, rs_a).T_0 + (rem\_req_1, rmrq_1).T_1 + (rem\_req_2, rmrq_2).T_2 + (rem\_req_3, rmrq_3).T_3;$$
$$\#T_0 = (rem\_req_1, rmrq_1).T_{01} + (rem\_req_2, rmrq_2).T_{02} + (rem\_req_3, rmrq_3).T_{03};$$
$$\#T_1 = (tm2ccu, rs_a).T_{01} + (rem\_req_2, rmrq_2).T_{12} + (rem\_req_3, rmrq_3).T_{13};$$
$$\#T_2 = (tm2ccu, rs_a).T_{02} + (rem\_req_1, rmrq_1).T_{12} + (rem\_req_3, rmrq_3).T_{23};$$
$$\#T_3 = (tm2ccu, rs_a).T_{13} + (rem\_req_1, rmrq_1).T_{13} + (rem\_req_2, rmrq_2).T_{23};$$
$$\#T_{01} = (rem\_req_2, rmreq_2).(rem\_req_3, rmreq_3).TM_0a + (rem\_req_3, rmreq_3).(rem\_req_2, rmreq_2).TM_0a;$$
$$\#T_{02} = (rem\_req_1, rmreq_1).(rem\_req_3, rmreq_3).TM_0a + (rem\_req_3, rmreq_3).(rem\_req_1, rmreq_1).TM_0a;$$
$$\#T_{03} = (rem\_req_1, rmreq_1).(rem\_req_2, rmreq_2).TM_0a + (rem\_req_2, rmreq_2).(rem\_req_1, rmreq_1).TM_0a;$$
$$\#T_{12} = (tm2ccu, rs_a).(rem\_req_3, rmreq_3).TM_0a + (rem\_req_3, rmreq_3).(tm2ccu, rs_a).TM_0a;$$
$$\#T_{13} = (tm2ccu, rs_a).(rem\_req_2, rmreq_2).TM_0a + (rem\_req_2, rmreq_2).(tm2ccu, rs_a).TM_0a;$$
$$\#T_{23} = (tm2ccu, rs_a).(rem\_req_1, rmreq_1).TM_0a + (rem\_req_1, rmreq_1).(tm2ccu, rs_a).TM_0a;$$

$$\#TM_0b = (ccu2tm, rs_b).M_0 + (rem\_reply_1, rmrp_1).M_1 + (rem\_reply_2, rmrp_2).M_2 + (rem\_reply_3, rmrp_3).M_3;$$
$$\#M_0 = (rem\_reply_1, rmrp_1).M_{01} + (rem\_reply_2, rmrp_2).M_{02} + (rem\_reply_3, rmrp_3).M_{03};$$
$$\#M_1 = (ccu2tm, rs_b).M_{01} + (rem\_reply_2, rmrp_2).M_{12} + (rem\_reply_3, rmrp_3).M_{13};$$
$$\#M_2 = (ccu2tm, rs_b).M_{02} + (rem\_reply_1, rmrp_1).M_{12} + (rem\_reply_3, rmrp_3).M_{23};$$
$$\#M_3 = (ccu2tm, rs_b).M_{13} + (rem\_reply_1, rmrp_1).M_{13} + (rem\_reply_2, rmrp_2).M_{23};$$
$$\#M_{01} = (rem\_reply_2, rmrp_2).(rem\_reply_3, rmrp_3).TM'_0b + (rem\_reply_3, rmrp_3).(rem\_reply_2, rmrp_2).TM'_0b;$$

$$\#M_{02} = (rem\_reply_1, rmrp_1).(rem\_reply_3, rmrp_3).TM'_0b + (rem\_reply_3, rmrp_3).(rem\_reply_1, rmrp_1).TM'_0b;$$
$$\#M_{03} = (rem\_reply_1, rmrp_1).(rem\_reply_2, rmrp_2).TM'_0b + (rem\_reply_2, rmrp_2).(rem\_reply_1, rmrp_1).TM'_0b;$$
$$\#M_{12} = (ccu2tm, rs_b).(rem\_reply_3, rmrp_3).TM'_0b + (rem\_reply_3, rmrp_3).(ccu2tm, rs_b).TM'_0b;$$
$$\#M_{13} = (ccu2tm, rs_b).(rem\_reply_2, rmrp_2).TM'_0b + (rem\_reply_2, rmrp_2).(ccu2tm, rs_b).TM'_0b;$$
$$\#M_{23} = (ccu2tm, rs_b).(rem\_reply_1, rmrp_1).TM'_0b + (rem\_reply_1, rmrp_1).(ccu2tm, rs_b).TM'_0b;$$
$$\#TM'_0b = (reply, r\_reply).TM_0b;$$

$$\#TM_0 = TM_0a <> TM_0b;$$

The state space of this PEPA model (referred to as *ModelD*) is considerably larger than *ModelC* in the previous chapter. In order to evaluate the system, the best solution for this model is to follow the decompositional evaluation approach.

6.1.2 Decomposing *ModelD*

ModelD can be decomposed into four submodels each consisting of a PE. Each submodel *PE* can be further decomposed into component submodels. The decomposition of *ModelD* also takes advantage of the existing PEPA specification from the previous chapter. Except for submodels TM_0a and TM_0b, the other submodels can reuse the specifications of the submodels of decomposed *ModelC*.

The specification of submodel TM_0a can be transformed directly from component TM_0a of *ModelD*. However, the specification of submodel TM_0b requires some attention. The responsibility of submodel TM_0b is to ensure that TM_0 receives the replies from all four components (CCU_0, TM_1, TM_2 and TM_3) before committing. For this reason, four queue generators are included in submodel TM_0b, each of which represents the arrival of replies from one of the submodels CCU_0c, TM_1b, TM_2b and TM_3b, respectively. To optimise the system performance, these replies may arrive in any order whenever the four components which supply

the replies are ready provided that TM_0 receives the replies from all four components (**Figure 6.2(a)**). The specifications of submodel TM_0b can be expressed as follows:

$$\#Q_{00} = (arrival, lambda_0).Q_{01};$$
$$\#Q_{0n} = (arrival, lambda_0).Q_{0(n+1)} + (ccu2tm, infty).Q_{0(n-1)};$$
$$\#Q_{0N} = (ccu2tm, infty).Q_{0(N-1)};$$

$$\#Q_{i0} = (arrival, lambda_i).Q_{i1};$$
$$\#Q_{in} = (arrival, lambda_i).Q_{i(n+1)} + (rem\_reply_i, infty).Q_{i(n-1)};$$
$$\#Q_{iN} = (rem\_reply_i, infty).Q_{i(N-1)};$$

$$where\ N \in \Gamma^+;\ i \in \{1,2,3\};\ n \in \{1..N-1\}.$$

$$\#TM_0b = (ccu2tm, rs_b).M_0 + (rem\_reply_1, rmrp_1).M_1 + (rem\_reply_2, rmrp_2).M_2 + (rem\_reply_3, rmrp_3).M_3;$$
$$\#M_0 = (rem\_reply_1, rmrp_1).M_{01} + (rem\_reply_2, rmrp_2).M_{02} + (rem\_reply_3, rmrp_3).M_{03};$$
$$\#M_1 = (ccu2tm, rs_b).M_{01} + (rem\_reply_2, rmrp_2).M_{12} + (rem\_reply_3, rmrp_3).M_{13};$$
$$\#M_2 = (ccu2tm, rs_b).M_{02} + (rem\_reply_1, rmrp_1).M_{12} + (rem\_reply_3, rmrp_3).M_{23};$$
$$\#M_3 = (ccu2tm, rs_b).M_{13} + (rem\_reply_1, rmrp_1).M_{13} + (rem\_reply_2, rmrp_2).M_{23};$$
$$\#M_{01} = (rem\_reply_2, rmrp_2).(rem\_reply_3, rmrp_3).TM'_0b + (rem\_reply_3, rmrp_3).(rem\_reply_2, rmrp_2).TM'_0b;$$
$$\#M_{02} = (rem\_reply_1, rmrp_1).(rem\_reply_3, rmrp_3).TM'_0b + (rem\_reply_3, rmrp_3).(rem\_reply_1, rmrp_1).TM'_0b;$$
$$\#M_{03} = (rem\_reply_1, rmrp_1).(rem\_reply_2, rmrp_2).TM'_0b + (rem\_reply_2, rmrp_2).(rem\_reply_1, rmrp_1).TM'_0b;$$
$$\#M_{12} = (ccu2tm, rs_b).(rem\_reply_3, rmrp_3).TM'_0b + (rem\_reply_3, rmrp_3).(ccu2tm, rs_b).TM'_0b;$$
$$\#M_{13} = (ccu2tm, rs_b).(rem\_reply_2, rmrp_2).TM'_0b + (rem\_reply_2, rmrp_2).(ccu2tm, rs_b).TM'_0b;$$
$$\#M_{23} = (ccu2tm, rs_b).(rem\_reply_1, rmrp_1).TM'_0b + (rem\_reply_1, rmrp_1).(ccu2tm, rs_b).TM'_0b;$$
$$\#TM'_0b = (reply, r\_reply).TM_0b;$$

$$\#Model = TM_0b \langle ccu2tm, rem\_reply_1, rem\_reply_2, rem\_reply_3 \rangle (Q_{00} \langle\rangle Q_{10} \langle\rangle Q_{20} \langle\rangle Q_{30});$$

Submodel TM_0b has approximately 20000 states. This is a huge state-space and will take a long time to converge on a solution for the performance evaluation of the submodel. In order to overcome this problem, the idea of intermediate threading is presented.

6.1.3 Intermediate Threading

The idea behind this technique is to exploit the underlying parallelism among the activities within a submodel by introducing some intermediate processes at appropriate places. Consider again the situation in submodel TM_0b. Instead of handling all the replies directly from the four components **(Figure 6.2(a))**, submodel TM_0b can have an intermediate process which will handle the replies from three components in the same way that component TM_0b handles the replies. The intermediate process will inform component TM_0b after it has collected all three replies as required. As for component TM_0b, it now only has to handle the replies from the intermediate process and component CCU_0c **(Figure 6.2(b))**. Alternatively, submodel TM_0b can have two intermediate processes, each handling the replies from two components independently and simultaneously. Each of these intermediate processes will inform component TM_0b after it has completed collecting the replies from the components. Component TM_0b now only has to handle the replies from the intermediate processes **(Figure 6.2(c))**.

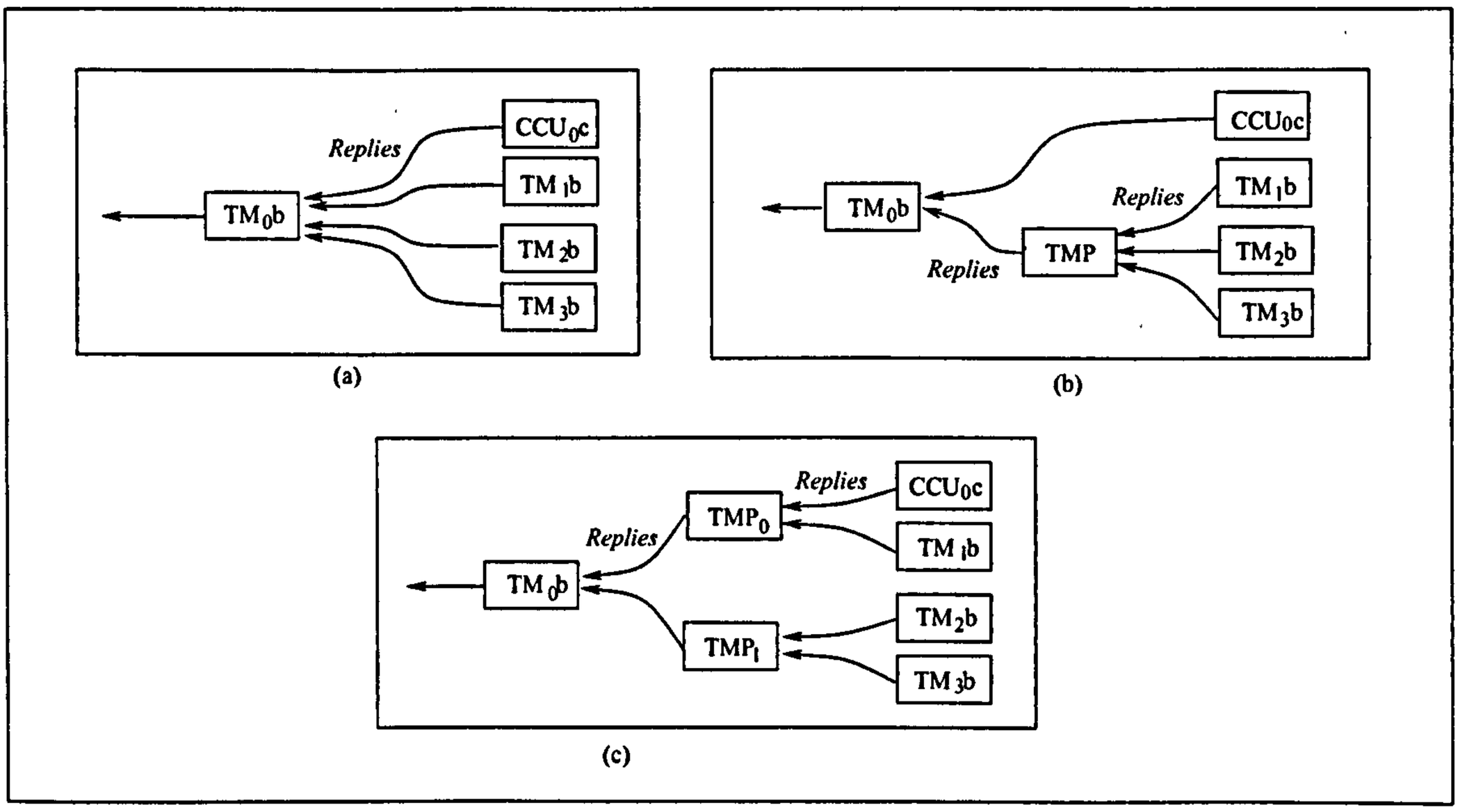


Figure 6.2. (a) Component TM_0b receives replies from four components.
(b) Component TM_0b with one intermediate process (*TMP*).
(c) Component TM_0b with two intermediate processes (TMP_0 and TMP_1).

Component TM_0b in **Figure 6.2 (b)** and **(c)** has included the intermediate processes and this allows some activities to be performed simultaneously. Nevertheless, this still ensures that TM_0 receives all four replies before proceeding with any other activity. Besides, the inclusion of the intermediate processes also allows submodel TM_0b to be decomposed further into smaller models. For example, submodel TM_0b illustrated in **Figure 6.2(b)** can be broken down into two parts: the first part contains the activities of the intermediate process *TMP*, and the second consists of the activities of component TM_0b. On the other hand, submodel TM_0b, illustrated in **Figure 6.2(c)**, can be decomposed into three smaller models, two of which consist of the intermediate processes *TMP0* and *TMP1*, and the third contains component TM_0b.

With this further decomposed submodel TM_0b, system performance evaluation can be conducted following the decompositional evaluation approach. Both choices of the decomposed submodel TM_0b illustrated in **Figure 6.2 (b)** and **(c)** are studied in the experiment.

Figure 6.3 summarises all four sets of experiment results for four different transaction arrival rates. Each set of experiments is represented by two graphs. Each graph represents the system performance of *ModelD* obtained by using the decompositional evaluation approach in which submodel TM_0b has been further decomposed into two (Choice A) or three smaller models (Choice B).

There are eight performance graphs shown in **Figure 6.3**. However, most of them are overlapping. This is because there is little noticeable difference between the graphs for Choice A and Choice B. In others words, although submodel TM_0b has been decomposed into two or three smaller models according to the number of intermediate processes included, this has little effect on the overall system performance.

In the subsequent case study, the intermediate threading technique will be applied on top of the decompositional evaluation approach to solve the system performance evaluation when necessary.

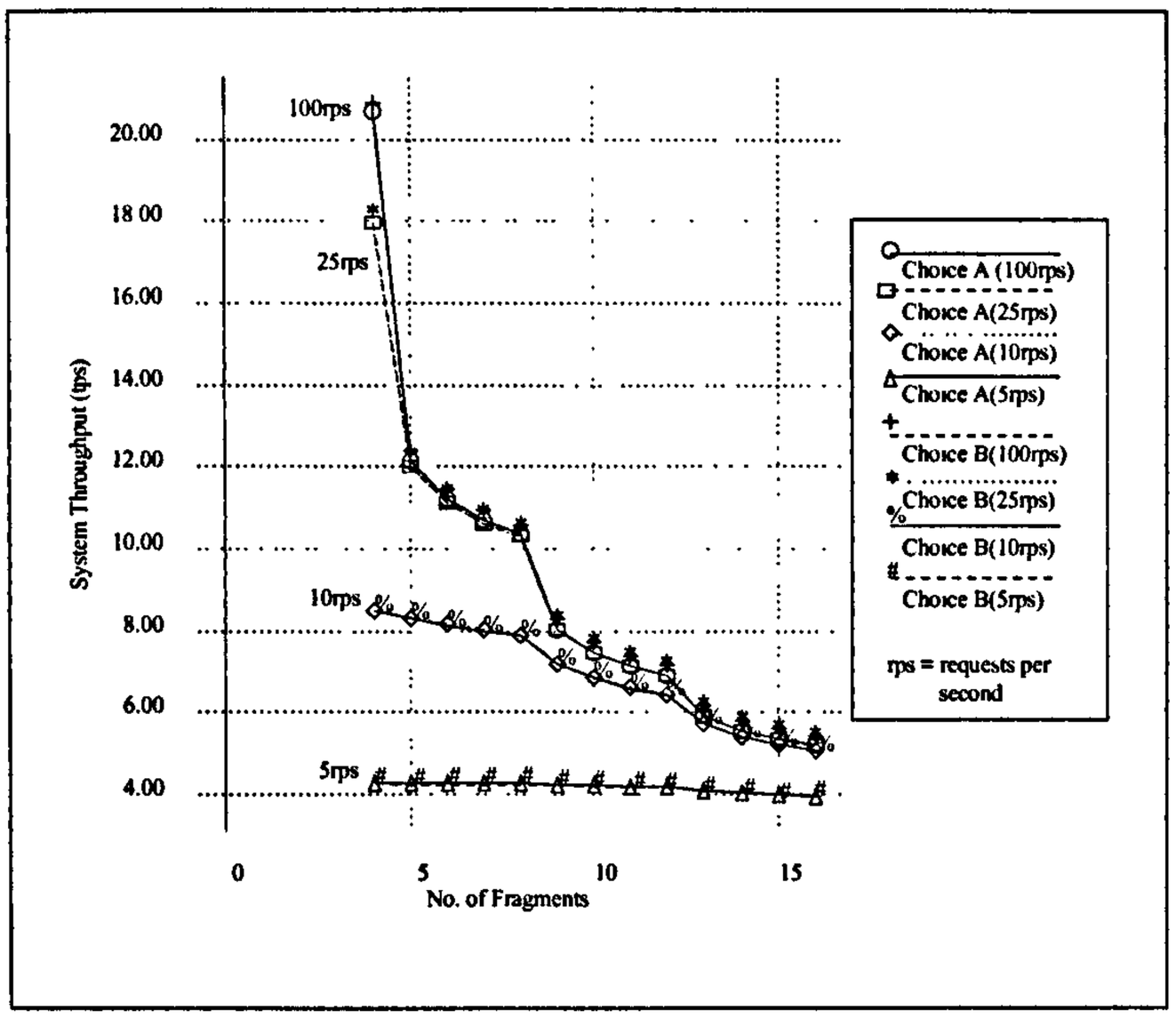


Figure 6.3. Experimental results obtained using Choice A and B for various arrival rates.

6.2 System Performance and Scalability

The system is further extended from 4 PEs to 8 PEs and 16 PEs, with a disk attached to each PE. The model for the cases of 8 and 16 PEs are constructed in the same way as for *ModelD*, and are evaluated according to the decompositional evaluation approach. A series of experiments have been conducted for these two models, together with *ModelD* (for 4 PE case) and *ModelC* (for 2 PE case), to study the influence of varying the number of PEs on the overall system performance.

The database size varies from 2 to 64 fragments (each fragment consisting of a page), while the transaction request arrival rate is set to 100. These fragments are distributed across the PEs following a round-robin data placement strategy. The results are summarised in **Figure 6.4.**

When the number of PEs is greater than the total number of fragments in the database of the system, the PEs that have no fragment allocated to them will not affect the system performance. Nevertheless, all four graphs show a step-like behaviour as the number of fragments in the database varies from 2 fragments to 64 fragments. The throughput of each configuration declines

gradually as the database size increases except at certain points where sharp drops are observed. This happens at each point where the number of fragments is a multiple of the number of PEs in the system. When the number of fragments is an exact multiple of the number of PEs in the system, the database is distributed equally amongst the PEs and the load is balanced. When adding an extra fragment to the system, one of the PE will have one more fragment than other PEs and will become the bottleneck and subsequently affect the overall system performance. The overall throughput of the system with 16 PEs is approximately twice the throughput of the system with 8 PEs. It is about four times the throughput of the system with 4 PEs and nearly eight times that of the system with 2 PEs.

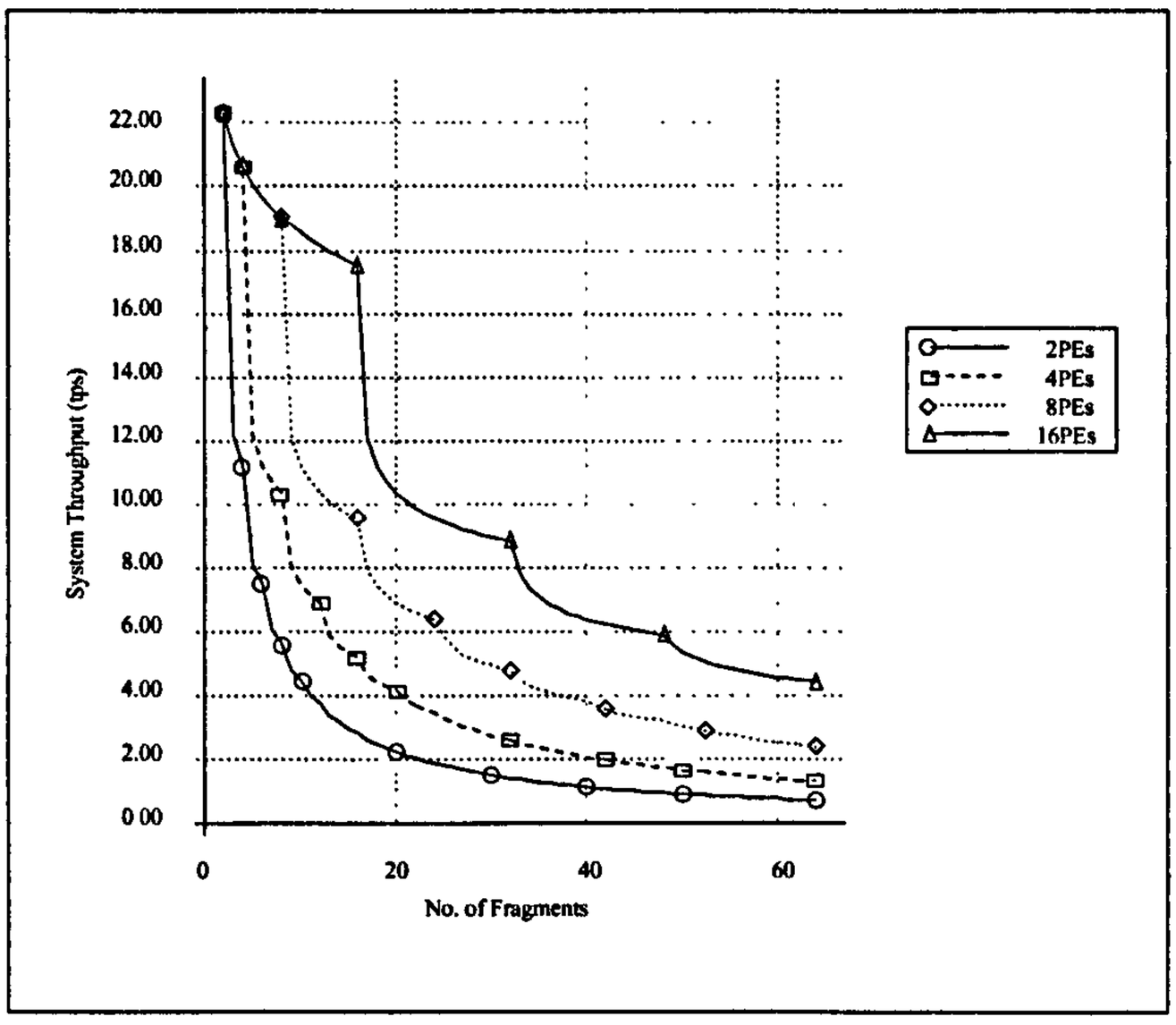


Figure 6.4. The throughputs (tps) of models with multiple nodes with single disk each.

6.3 Summary

This chapter introduces the idea of intermediate threading and further decomposition for submodels through the example of a simple database system running on a shared nothing parallel platform with 4 PEs. This technique is important especially for a decomposed model that

consists of submodels that have a huge state-space and may be unable to converge on a solution. This technique is used in the subsequent case study on top of the decomposition evaluation approach where necessary.

This chapter concludes with a study comparing the performance of systems with various numbers of nodes and has shown how PEPA can be used to model performance in such cases.

Chapter 7

MODELLING PARALLEL TRANSACTION PROCESSING WITH PEPA

7.0 Introduction

In this chapter, PEPA is used to develop models based on actual parallel database systems. First, it is used to model a parallel database system that utilises only inter-transaction parallelism. Then it is extended to model a system that utilises both inter-transaction and intra-transaction parallelism. Both systems run on the ICL Goldrush platform and are viewed in the context of the TPC-B transaction processing benchmark.

The following section briefly discusses the Goldrush platform and the TPC-B benchmark configured to suit the study. Section 7.2 investigates a model that based on the INGRES Cluster DBMS running on Goldrush while section 7.3 discusses how PEPA is used to model the Informix XPS DBMS, also running on Goldrush.

7.1 The Platform and Benchmark

The ICL Goldrush [107] is a shared-nothing parallel machine. It consists of a set of processing elements (PEs), each of which runs its own copy of the operating system, and a high speed DeltaNet interconnect that acts as the communication medium between Goldrush elements. The machine can support up to 64 PEs and each PE can be connected to up to 12 disks. However, for the purpose of this study, each PE is assumed to have only one disk attached to it and the number of PEs varies from 1 to 12.

The TPC-B transaction processing benchmark is a standard benchmark that is used to evaluate the performance of a DBMS in terms of transaction throughput and response time. However, this study only considers transaction throughput. For this study, it is assumed that there are 20 branches, each with 10 tellers, one hundred thousand accounts and a history relation that is large enough to hold all history records generated (approximately 864000 rows). The size of each tuple in relation *Branch*, *Teller* and *Account* is 100 bytes whereas in relation *History*, tuples are 50 bytes in length. Each relation is partitioned into 20 fragments (i.e. the number of branches in the database) according to the hash partitioning strategy. These fragments are distributed across the PEs according to the size data placement strategy. In this case, the data placement starts the distribution with relation *History*, the largest relation in the database. Each fragment of the relation *History* is allocated to the PE that has the most space available. The data placement continues with the fragments of relation *Account* once the fragments of relation *History* are exhausted, and then with the fragments of relations *Teller* and *Branch* respectively.

7.2 Modelling Ingres Cluster DBMS with PEPA

In this section, PEPA is used to develop a model that is based on the inter-transaction parallel processing feature of Goldrush Ingres Cluster DBMS [113] running on a shared-nothing parallel platform, in this case ICL Goldrush. Although the Ingres Cluster DBMS consists of many components (such as DBMS servers, Communication Servers, Recovery Process and so on), the system has been simplified to suit the purpose of this study that focuses on the utilisation of inter-transaction parallelism. For this reason, it is assumed that every PE has a DBMS server which consists of one transaction manager, one concurrency control unit, one lock manager and one buffer manager (**Figure 7.1**).

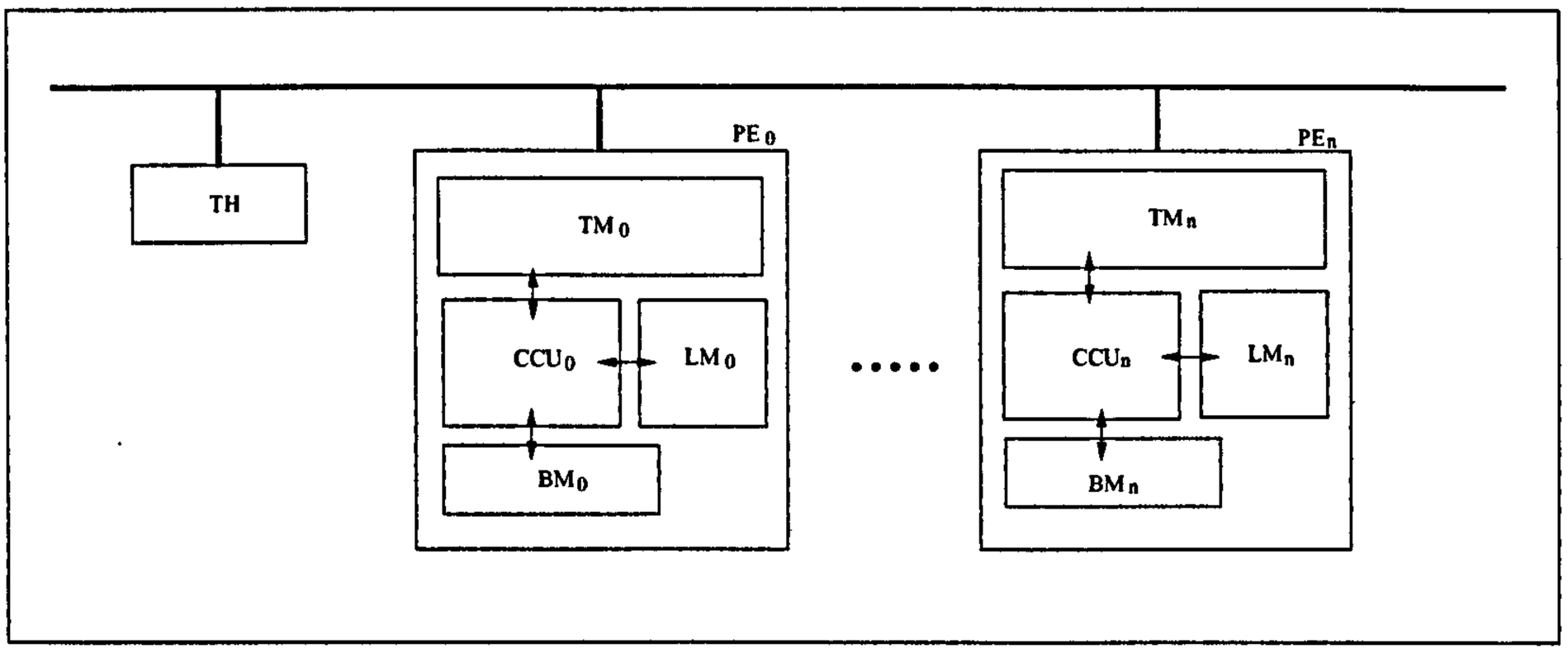


Figure 7.1. Structure of simplified Ingres Cluster DBMS.

The system supports the TPC-B benchmark that requires it to perform a series of update operations on the database. For every update, it is assumed that the updated data page is immediately written back to the disk. It takes 41.11 msec to read a page from the disk and 42.0 msec to write a page to the disk (based on the figures from [111]). No cacheing is considered in this study although locking is considered. It is also assumed that the relations are appropriately indexed so that the system need not search the entire database for a matched tuple.

7.2.1 Transaction Handling

It is assumed that there is a transaction host (TH) whose job is to handle transaction requests from the users. When the TH receives a request from a user, it sends a request to a PE that contains the data (i.e. the fragment of relation Branch that matches the Branch ID of the condition in the query). The TH waits for the reply from the PE while new transaction requests may arrive in the interim.

When a PE receives a transaction request from the TH, the transaction manager (TM_i) on the PE will forward the request to the local concurrency control unit (CCU_i) if all the data requested are stored on the disk associated with the PE. Otherwise, the TM_i has to send one or more remote requests to other PE(s) that contain the required data fragment(s) besides sending a local request

to the CCU_i. If all the data are stored on the same disk, the TM_i just waits for the reply from the CCU_i and commits the transaction after that. Otherwise, the TM_i will have to wait for the remote replies from other PE(s) in addition to the reply from the CCU_i before committing a transaction and sends a message to the TH. Meanwhile, new transaction requests may arrive at the PE and will be dealt with in the same way.

The TM_i may also receive remote requests from other PEs. In this case, the TM_i just forwards the remote request as a local request to the CCU_i. However, when the CCU_i returns the result of the remote request, the TM_i replies directly to the PE(s) that sent the remote requests earlier instead of relaying the result to the TH.

The function of the CCU_i is similar to those discussed in the previous chapters. However, the LM_i has a more complex task to handle in this case. The LM_i is assumed to follow the strict two-phase locking protocols. When the LM_i receives a request from the CCU_i, it has to ensure that all data pages required are available so that it can lock these pages exclusively for the request. If any of these pages is currently locked by another request, the new request will be blocked waiting in the queue. When a request has completed, the LM_i will release all the locks held by the request immediately. If there is a request waiting for the page(s) in the queue, the request will be granted and the data page(s) will be locked.

The BM_i waits for the requests from the CCU_i. A request from the CCU_i may require the BM_i to update three relations (Account, Branch and Teller) and insert a record into relation History. The request may also require the BM_i to perform only one of the four operations stated in the TPC-B transaction query. In other words, the data manipulation is dependent on the data stored on the disk. At the end of the process, the BM_i returns a message to the CCU_i.

7.2.2 Representing the System with PEPA

In general, the PEPA model can be viewed as consisting of one component *TH* and several homogeneous component *PEs* communicating with each other. However, for the purpose of this study, the number of PEs varies between 1 and 12. This produces different data distribution patterns as the system configuration changes and consequently different system behaviours. For this reason, different PEPA models are required for different cases.

Although the PEPA models may be different from each other, there are components in the models that are less affected by the system configuration. They are components CCU_i and LM_i of the homogeneous PEs of each PEPA model.

The specification of component CCU_i is similar to those in the previous examples (e.g. *ModelA*) in the previous chapters. Hence, the PEPA specification of component *CCU* of *ModelA*, for instance, can be reused for component CCU_i here. However, component LM_i has a more complex task now and is expressed as follows:

$$\#Q_0E_0 = (ccu_i2lm_i,\ infty).Q_0E_{0\wedge};$$
$$\#Q_0E_{0\wedge} = (lm_i2ccu_i,\ rs).Q_0E_1;$$
$$\#Q_0E_n = (ccu_i2lm_i,\ infty).Q_0E_{n\wedge} + (dequeue,\ infty).Q_0E_{n-1};$$
$$\#Q_0E_{m\wedge} = (lm_i2ccu_i,\ rs).Q_0E_{m+1} + (queue,\ rq).Q_1E_m + (dequeue,\ infty).Q_0E_{(m-1)\wedge};$$
$$\#Q_0E_{N\wedge} = (queue,\ rq).Q_1E_N + (dequeue,\ infty).Q_0E_{(N-1)\wedge};$$

$$\#Q_wE_0 = (lm_i2ccu_i,\ rs).Q_{(w-1)}E_1;$$
$$\#Q_wE_{0\wedge} = (lm_i2ccu_i,\ rs).Q_{(w-1)}E_{1\wedge};$$
$$\#Q_wE_1 = (ccu_i2lm_i,\ infty).Q_wE_{1\wedge} + (dequeue,\ infty).Q_wE_0;$$
$$\#Q_vE_{1\wedge} = (lm_i2ccu_i,\ rs).Q_vE_2 + (queue,\ rq).Q_{(v+1)}E_1 + (dequeue,\ infty).Q_vE_{0\wedge};$$
$$\#Q_XE_{1\wedge} = (lm_i2ccu_i,\ rs).Q_XE_2 + (dequeue,\ infty).Q_XE_{0\wedge};$$

$$\#Q_wE'_m = (lm_i2ccu_i,\ rs).Q_{(w-1)}E_{(m+1)};$$
$$\#Q_wE'_{m\wedge} = (lm_i2ccu_i,\ rs).Q_{(w-1)}E_{(m+1)\wedge};$$

$$\#Q_wE_p = (ccu_i2lm_i,\ infty).Q_wE_{p\wedge} + (dequeue,\ infty).Q_wE_{(p-1)} + (dequeue,\ infty).Q_wE'_{(p-1)};$$
$$\#Q_vE_{k\wedge} = (lm_i2ccu_i,\ infty).Q_vE_{(k+1)} + (queue,\ rq).Q_{(v+1)}E_k + (dequeue,\ infty).Q_vE_{(k-1)\wedge} + (dequeue,\ infty).Q_vE'_{(k-1)\wedge};$$
$$\#Q_vE_{N\wedge} = (queue,\ rq).Q_{(v+1)}E_N + (dequeue,\ infty).Q_vE_{(N-1)\wedge} + (dequeue,\ infty).Q_vE'_{(N-1)\wedge};$$

$$\#Q_XE_{k\wedge} = (lm_i2ccu_i,\ rs).Q_XE_{(k+1)} + (dequeue,\ infty).Q_XE_{(k-1)\wedge} + (dequeue,\ infty).Q_XE'_{(k-1)\wedge};$$

$$\#Q_XE_{N\wedge} = (dequeue,\ infty).Q_XE_{(N-1)\wedge} + (dequeue,\ infty).Q_XE'_{(N-1)\wedge};$$

$$\#LMb = (release, infty).(dequeue, rs).LMb;$$

$$\#LM = LMb \langle dequeue \rangle Q_0E_0;$$

$$where \quad n \in \{1..N\};\ w \in \{1..X\};\ m \in \{1..N\text{-}1\};\ v \in \{1..X\text{-}1\};$$
$$p \in \{2..N\};\ k \in \{2..N\text{-}1\};\qquad N, X \in \Gamma^{+}$$

The specification of component LM_i above allows N requests to wait in the queue and another X requests to be granted and proceed with data manipulation. In contrast, the PEPA specification for components TH, TM_i and BM_i are significantly different that they are discussed in the next section under separate case studies.

7.2.3 Case Studies

A few cases are selected and discussed in the following subsections. These cases have different complexity and require different approaches.

A. 3 PE Case

Figure 7.2(a) shows the data distribution pattern of the relations (*Branch* (B), *Teller* (T), *Account* (A) and *History* (H)) amongst the 3 PEs. The numbers in the diagram represent the fragment IDs of each relation. In this case, the fragments of relation *Branch* are only allocated to PE_0 and PE_1. For this reason, the TH only sends the transaction requests to the PEs as shown in **Figure 7.2(b)**. Similarly, the TH waits for the replies from both TM_0 and TM_1.

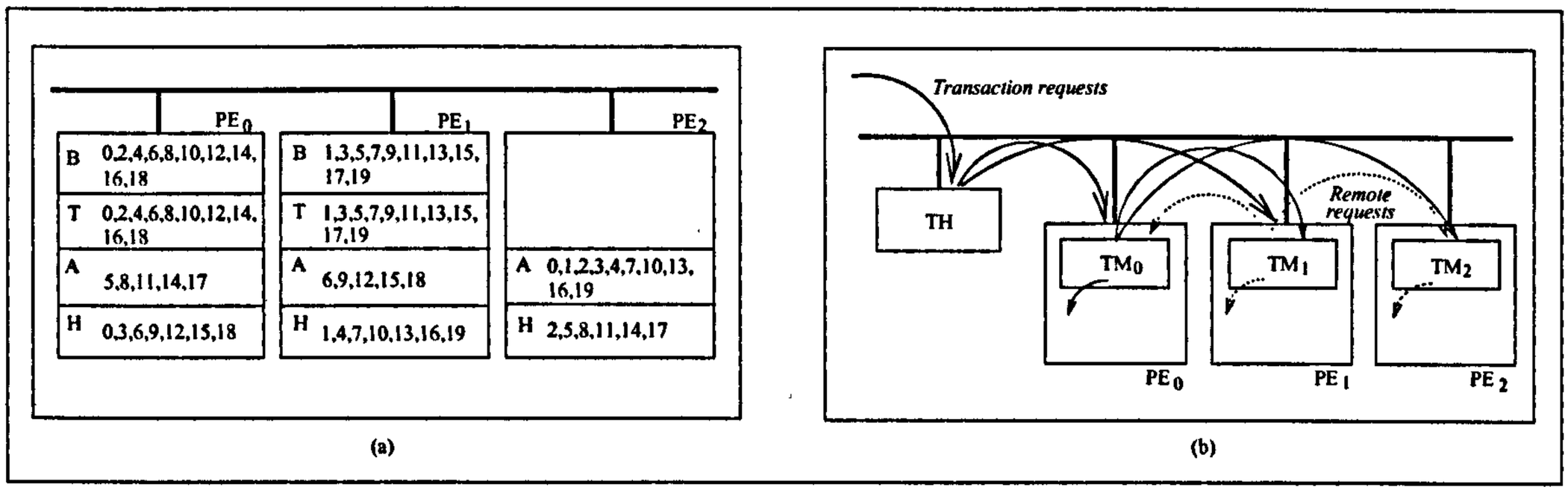


Figure 7.2. (a) Data placement for the 3 PE case.
(b) Transaction requests and remote requests distribution.

TM_0 waits for the transaction requests from the TH. When TM_0 receives a transaction request from the TH, it forwards one request to CCU_0 and one remote request to TM_1 or TM_2. Occasionally, TM_0 is required to forward one request to CCU_0 and one remote request each to TM_1 and TM_2. On the other hand, TM_0 also receives remote requests from TM_1 in which case it will forward the remote request as a local request to CCU_0. TM_0 waits for the replies from these components.

If TM_0 receives a reply from CCU_0, it needs to receive one remote reply from either or both of TM_1 or TM_2, before committing a transaction. There is also a possibility when the reply from CCU_0 is a reply to a remote request of TM_1, in which case TM_0 needs to send the result to TM_1 as a remote reply.

As for TM_1, it also waits for the transaction request from the TH before sending a request to CCU_1 and one remote request to either or both of TM_0 or TM_2. TM_1 also receives the remote requests from TM_0. TM_1 waits for the reply from CCU_1 or the remote replies from TM_0 and TM_2 before committing a transaction or replying a remote reply to TM_0.

TM_2 only receives the remote requests from both TM_0 and TM_1. When TM_2 receives a remote request from either TM_0 or TM_1, it sends a request to CCU_2 and waits for the reply before returning a message to either TM_0 or TM_1. The PEPA specification for components *TH*, TM_0, TM_1 and TM_2 can be expressed as follows,

$\#THa = (request, r\_req).TH'a;$
$\#TH'a = (request_0, rq_0).THa + (request_1, rq_1).THa;$
$\#THb = (reply_0, rp_0).TH'b + (reply_1, rp_1).TH'b;$
$\#TH'b = (reply, r\_reply).THb;$
$\#TH = THa <> THb;$

$\#TM_0a = (request_0, infty).(tm_02ccu_0, rs).TM'_0a +$
$(rmreq_{10}, infty).(tm_02ccu_0, rs).TM_0a;$
$\#TM'_0a = (rmreq_{01}, r_{01a}).TM_0a + (rmreq_{01}, r_{01b}).TM''_0a + (rmreq_{02}, r_{02a}).TM_0a;$
$\#TM''_0a = (rmreq_{02}, rs).TM_0a;$
$\#TM_0b = (ccu_02tm_0, infty).TM'_0b + (rmreply_{10}, infty).TM''_0b +$
$(rmreply_{20}, infty).TM^{\#}_0b;$
$\#TM'_0b = (rmreply_{01}, rs).TM_0b + (rmreply_{10}, infty).TM^{x}_0b +$
$(rmreply_{10}, infty).TM^{*}_0b + (rmreply_{20}, infty).TM^{x}_0b;$
$\#TM''_0b = (ccu_02tm_0, infty).TM^{x}_0b + (ccu_02tm_0, infty).TM^{*}_0b +$
$(rmreply_{20}, infty).TM^{\wedge}_0b;$
$\#TM^{\#}_0b = (ccu_02tm_0, infty).TM^{x}_0b + (ccu_02tm_0, infty).TM^{+}_0b +$
$(rmreply_{10}, infty).TM^{\wedge}_0b;$
$\#TM^{*}_0b = (rmreply_{01}, rs).TM''_0b + (rmreply_{20}, infty).TM^{x}_0b;$
$\#TM^{\wedge}_0b = (ccu_02tm_0, infty).TM^{x}_0b;$
$\#TM^{+}_0b = (rmreply_{01}, rs).TM^{\#}_0b + (rmreply_{10}, infty).TM^{x}_0b;$
$\#TM^{x}_0b = (reply_0, rs).TM_0b;$
$\#TM_0 = TM_0a <> TM_0b;$

$\#TM_1a = (request_1, infty).(tm_12ccu_1, rs).TM'_1a +$
$(rmreq_{01}, infty).(tm_12ccu_1, rs).TM_1a;$
$\#TM'_1a = (rmreq_{10}, r_{10a}).TM_1a + (rmreq_{10}, r_{10b}).TM''_1a + (rmreq_{12}, r_{12a}).TM_1a;$
$\#TM''_1a = (rmreq_{12}, rs).TM_1a;$
$\#TM_1b = (ccu_12tm_1, infty).TM'_1b + (rmreply_{01}, infty).TM''_1b +$
$(rmreply_{21}, infty).TM^{\#}_1b;$
$\#TM'_1b = (rmreply_{10}, rs).TM_1b + (rmreply_{01}, infty).TM^{x}_1b +$
$(rmreply_{01}, infty).TM^{*}_1b + (rmreply_{21}, infty).TM^{x}_1b;$
$\#TM''_1b = (ccu_12tm_1, infty).TM^{x}_1b + (ccu_12tm_1, infty).TM^{*}_1b +$
$(rmreply_{21}, infty).TM^{\wedge}_1b;$
$\#TM^{\#}_1b = (ccu_12tm_1, infty).TM^{x}_1b + (ccu_12tm_1, infty).TM^{+}_1b +$
$(rmreply_{01}, infty).TM^{\wedge}_1b;$
$\#TM^{*}_1b = (rmreply_{10}, rs).TM''_1b + (rmreply_{21}, infty).TM^{x}_1b;$
$\#TM^{\wedge}_1b = (ccu_12tm_1, infty).TM^{x}_1b;$
$\#TM^{+}_1b = (rmreply_{01}, infty).TM^{x}_1b + (rmreply_{10}, rs).TM^{\#}_1b;$
$\#TM^{x}_1b = (reply_1, rs).TM_1b;$
$\#TM_1 = TM_1a <> TM_1b;$

$\#TM_2a = (rmreq_{02}, infty).(tm_22ccu_2, rs).TM_2a +$
$(rmreq_{12}, infty).(tm_22ccu_2, rs).TM_2a;$
$\#TM_2b = (ccu_22tm_2, infty).TM'_2b;$
$\#TM'_2b = (rmreply_{20}, r_{20}).TM_2b + (rmreply_{21}, r_{21}).TM_2b;$
$\#TM_2 = TM_2a <> TM_2b;$

When BM_0 receives a request from CCU_0, it may be required to perform one of the following activities, depending on the data allocated to the disk:

1. Update relations *Branch*, *Teller* and *Account*;

2. Update relations *Branch* and *Teller*;

3. Update relations *Branch* and *Teller*, and insert into relation *History*;

4. Update relation *Account* only;

5. Insert into relation *History* only.

Likewise, BM_1 is also expected to perform one of the above activities. For this reason, both BM_0 and BM_1 can be expressed as follows,

$$
\begin{aligned}
&\#BM_i = (ccu_i2bm_i, infty).BM'_i;\\
&\#BM'_i = (manipulate, rm_0).BM''_i + (manipulate, rm_1).BM''_i +\\
&\qquad\qquad (manipulate, rm_2).BM''_i + (manipulate, rm_3).BM''_i +\\
&\qquad\qquad (manipulate, rm_4).BM''_i;\\
&\#BM''_i = (bm_i2ccu_i, rs).BM_i;\\
&where\ i \in \{0,1\}.
\end{aligned}
$$

The time taken to perform a transaction request is reflected in the activity rate rm_j of activity *(manipulate, rm_j)*. For instance, to update three relation pages, it takes 249.44 msec. This is equivalent to 4.01 tps.

In contrast, BM_2 has less choice of activities. When BM_2 receives a request from CCU_2, it may be required to:

1. Update relation *Account* and insert into relation *History*;

2. Update relation *Account* only;

3. Insert into relation *History* only.

So, BM_2 can be expressed as follows,

$$
\begin{aligned}
&\#BM_2 = (ccu_2bm_2, infty).BM'_2;\\
&\#BM'_2 = (manipulate, rm_0).BM''_2 + (manipulate, rm_1).BM''_2 +\\
&\qquad\qquad (manipulate, rm_2).BM''_2;\\
&\#BM''_2 = (bm_2ccu_2, rs).BM_2;
\end{aligned}
$$

Evaluating the PEPA model

The performance evaluation of this model follows the decompositional evaluation approach. The model is decomposed into submodels, each consisting of one or more atomic components. For instance, component TH is decomposed into submodels *THa* and *THb*, while the component *PEs* are each decomposed into submodels $TM_i a$, $TM_i b$, $CCU_i a$, and so on, as for *ModelA* in the previous example (*Chapter 5*).

The decompositional evaluation approach starts with submodel *THa*, followed by submodel $TM_0 a$ or submodel $TM_1 a$. The throughput of submodel *THa* becomes the arrival rate of the transaction requests in submodel $TM_0 a$ and submodel $TM_1 a$. The evaluation of submodel $TM_0 a$ depends not only on the arrival rate of transaction requests from the TH but also on the arrival rate of remote requests from TM_1 (submodel $TM_1 a$). Likewise, the evaluation of submodel $TM_1 a$ depends on both the arrival rate of transaction requests from the TH (submodel *THa*) and the arrival rate of remote requests from TM_0 (submodel $TM_0 a$). The recursive dependency between submodels $TM_0 a$ and $TM_1 a$ can cause the decompositional evaluation approach to iterate among these submodels without converging. For example, in order to evaluate submodel $TM_0 a$, a number is chosen arbitrarily for the arrival rate of remote requests from TM_1, which is unknown at the time. The throughput of submodel $TM_0 a$ is determined and becomes the remote request arrival rate for submodel $TM_1 a$. Submodel $TM_1 a$ is evaluated and produces a throughput. Submodel $TM_0 a$ is re-evaluated with the new arrival rate of remote requests taken from the throughput of submodel $TM_1 a$. Now, the throughput measured for submodel $TM_0 a$ may differ from the one estimated earlier. If this is the case, submodel $TM_1 a$ has to be re-evaluated and the whole process repeated, and may either converge on a solution or not.

One way to avoid this problem is to group these submodels into one lumped submodel. Therefore, submodels *THa*, $TM_0 a$, $TM_1 a$ and $TM_2 a$ are grouped together to form submodel *TMas* **(Figure 7.3(a))**. The PEPA model of submodel *TMas* is as follows:

$\#TM_0 = (request, r\_req).TM_0a$;
$\#TM_0a = (rmreq_0, r_0).TM_0 + (rmreq_1, r_1).TM_0$;

$\#TM_0x = (rmreq_0, infty).(tm2ccu_0, rs).TM_0b + (rmreq_{10}, infty).(tm2ccu_0, rs).TM_0x$;
$\#TM_0b = (rmreq_{01}, r_{01}a).TM_0x + (rmreq_{01}, r_{01}b).TM_0c + (rmreq_{02}, r_{02}a).TM_0x$;
$\#TM_0c = (rmreq_{02}, rs).TM_0x$;

$\#TM_1 = (rmreq_1, infty).TM_1a + (rmreq_{01}, infty).(tm2ccu_1, rs).TM_1$;
$\#TM_1a = (tm2ccu_1, rs).TM_1b$;
$\#TM_1b = (rmreq_{10}, r_{10}a).TM_1 + (rmreq_{10}, r_{10}b).TM_1c + (rmreq_{12}, r_{12}a).TM_1$;
$\#TM_1c = (rmreq_{12}, rs).TM_1$;

$\#TM_2 = (rmreq_{02}, infty).(tm2ccu_2, rs).TM_2 + (rmreq_{12}, infty).(tm2ccu_2, rs).TM_2$;

$\#TMas = TM_0 \langle rmreq_0, rmreq_1 \rangle$
$((TM_0x \langle rmreq_{10}, rmreq_{01} \rangle TM_1) \langle rmreq_{02}, rmreq_{12} \rangle TM_2)$;

The recursive dependency also occurs between submodel TM_0b and submodel TM_1b. For this reason, submodels THb, TM_0b, TM_1b and TM_2b are grouped together to form submodel $TMbs$ **(Figure 7.3(b))** and the PEPA model of this submodel is attached in **Appendix F.**

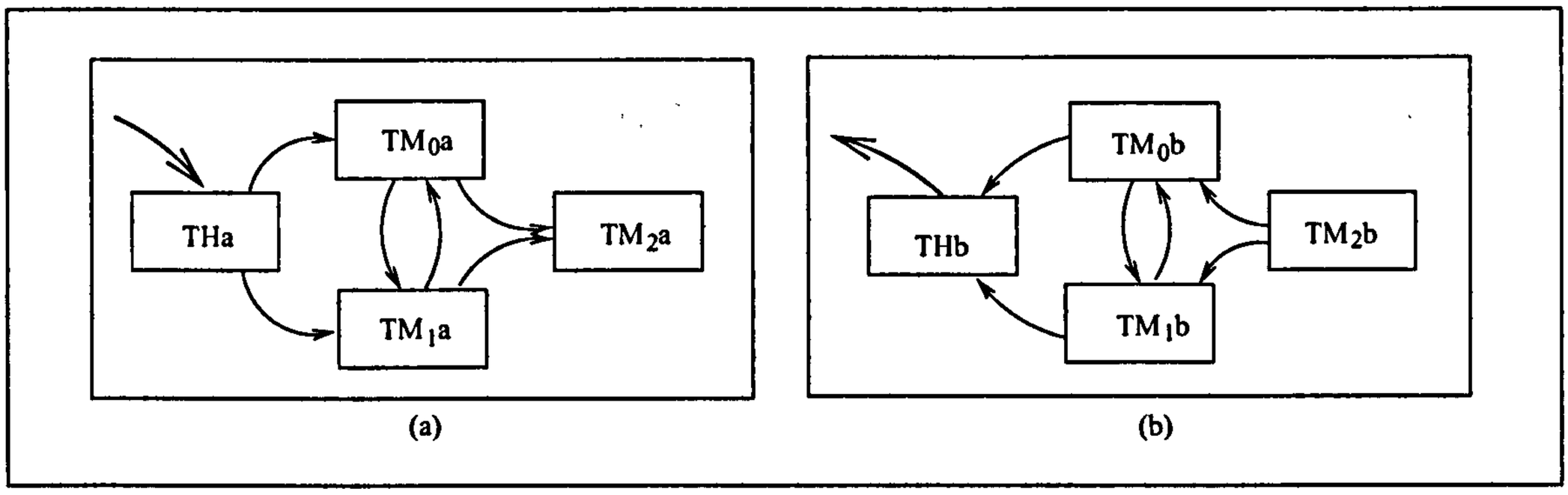


Figure 7.3. (a) Submodel TMas. (b) Submodel TMbs.

The recursive dependency also exists in each PE that involves submodels LM_ia, CCU_ib, CCU_ic, BM_i and LM_ib. In this case, submodel LM_ia needs feedback on dequeue requests from submodel LM_ib in order to complete the evaluation. On the other hand, the evaluation of submodel LM_ib depends of the throughput of submodel CCU_ic which in turn depends on the evaluation of submodel BM_i and so on **(Figure 7.4(a))**. As a result, submodel BCD_i is formed **(Figure 7.4(b))**. The decomposed model for the 3 PE case is evaluated and the throughput obtained is 8.17 tps.

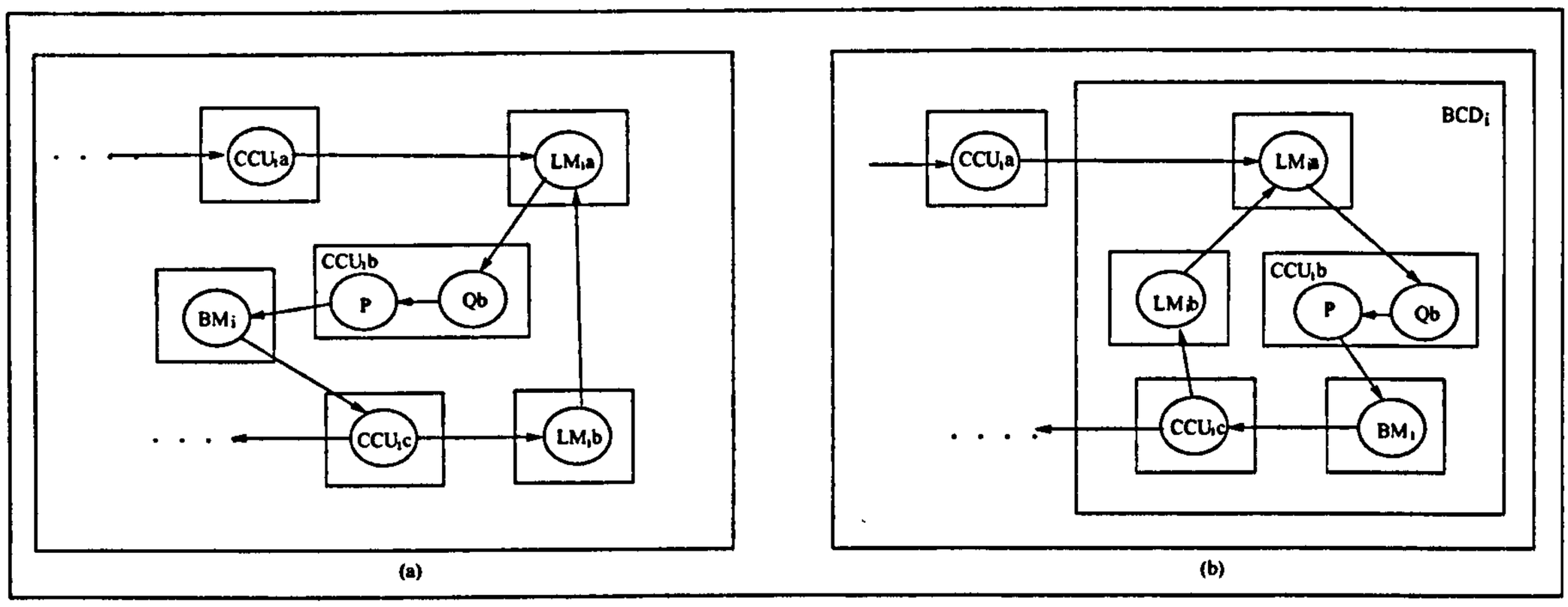


Figure 7.4. (a) Components of PE_i before lumping.
(b) Components of PE_i after lumping.

B. 5 PE Case

The case of 5 PEs is very straightforward because the relations are uniformly distributed across the PEs (**Figure 7.5(a)**). When the TH receives a request from a user, it just forwards the request to the PE that contains the data (**Figure 7.5(b)**). Since the related data is all stored on the same disk, the TM_i does not need to involve other TMs for transaction processing. When the TM_i receives a transaction request from the TH, it forwards the request to the CCU_i and waits for the reply. The responsibilities of components CCU_i and LM_i are straightforward and component BM_i has only one simple choice, which is to update three relations (*Branch*, *Teller* and *Account*) and insert a record into relation *History*.

The evaluation of this model, however, still follows the decompositional evaluation approach. It starts with submodel *THa*, followed by evaluating the submodels of each PE in turn.

The throughput estimated for this model is 17.15 tps.

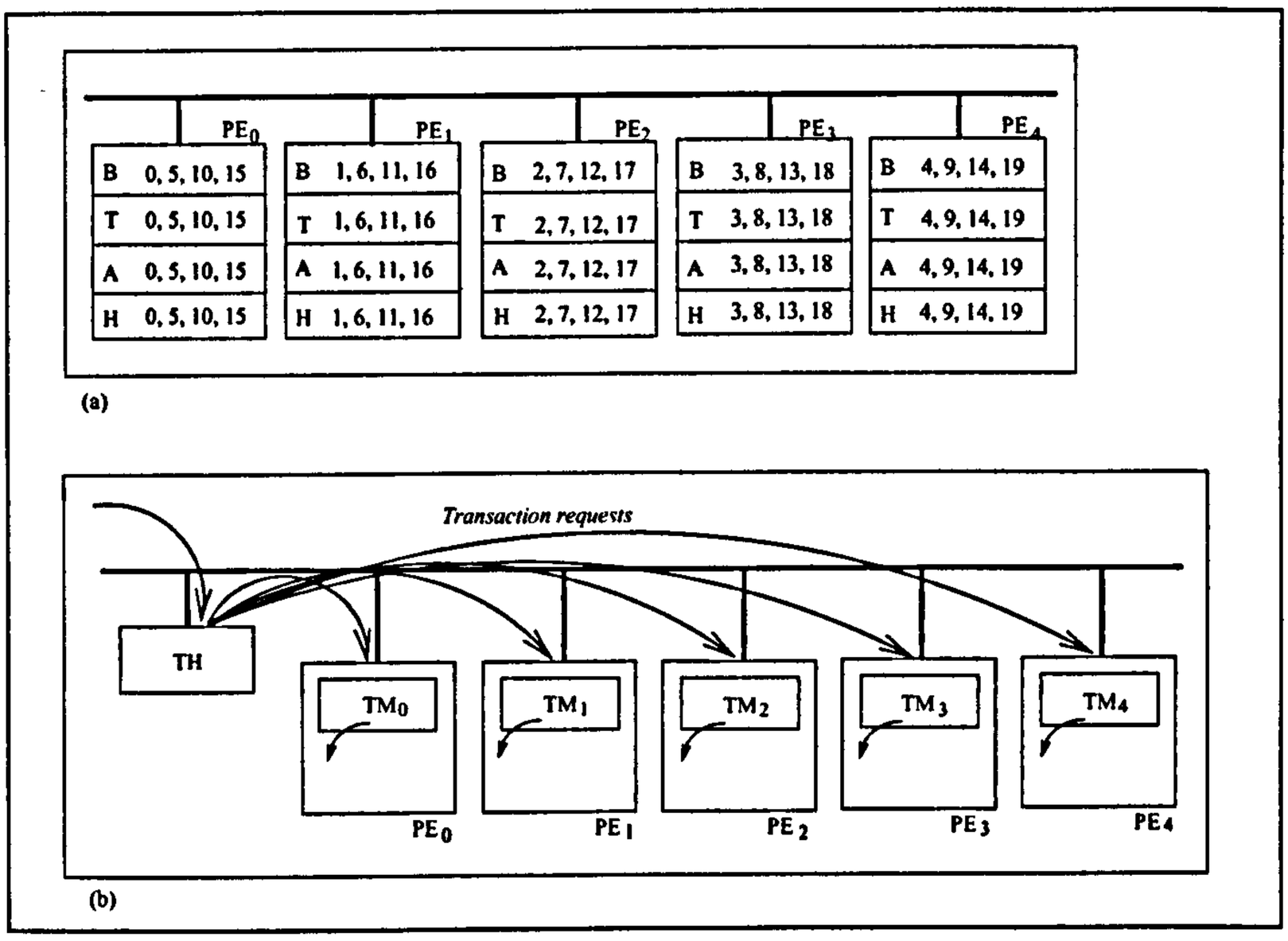


Figure 7.5. (a) Data placement of 5 PE case.
(b) Distribution of transaction requests.

C. 9 PE Case

As shown in **Figure 7.6(a)**, the fragments of relation *Branch* are all allocated to PE_8 only. For this reason, when the TH receives a request from a user, it simply forwards the request to TM_8 and waits for the reply (**Figure 7.6(b)**). Meanwhile, new transaction requests may arrive and will be dealt with.

When TM_8 receives a transaction request from the TH, it forwards a request to CCU_8 and one remote request to one of the TMs on PE_0 to PE_7, or it sends one remote request each to two of the TMs on other PEs. TM_8 waits for the reply from CCU_8 and one remote reply from another TM_i, or one remote reply each from two other TMs.

As for other TMs, each waits for a remote request from TM_8 and forwards the request to its CCU_i. When each of these TMs receives a reply from its CCU_i, it sends a remote reply to TM_8. The throughput estimated using the decompositional evaluation approach is 5.67 tps.

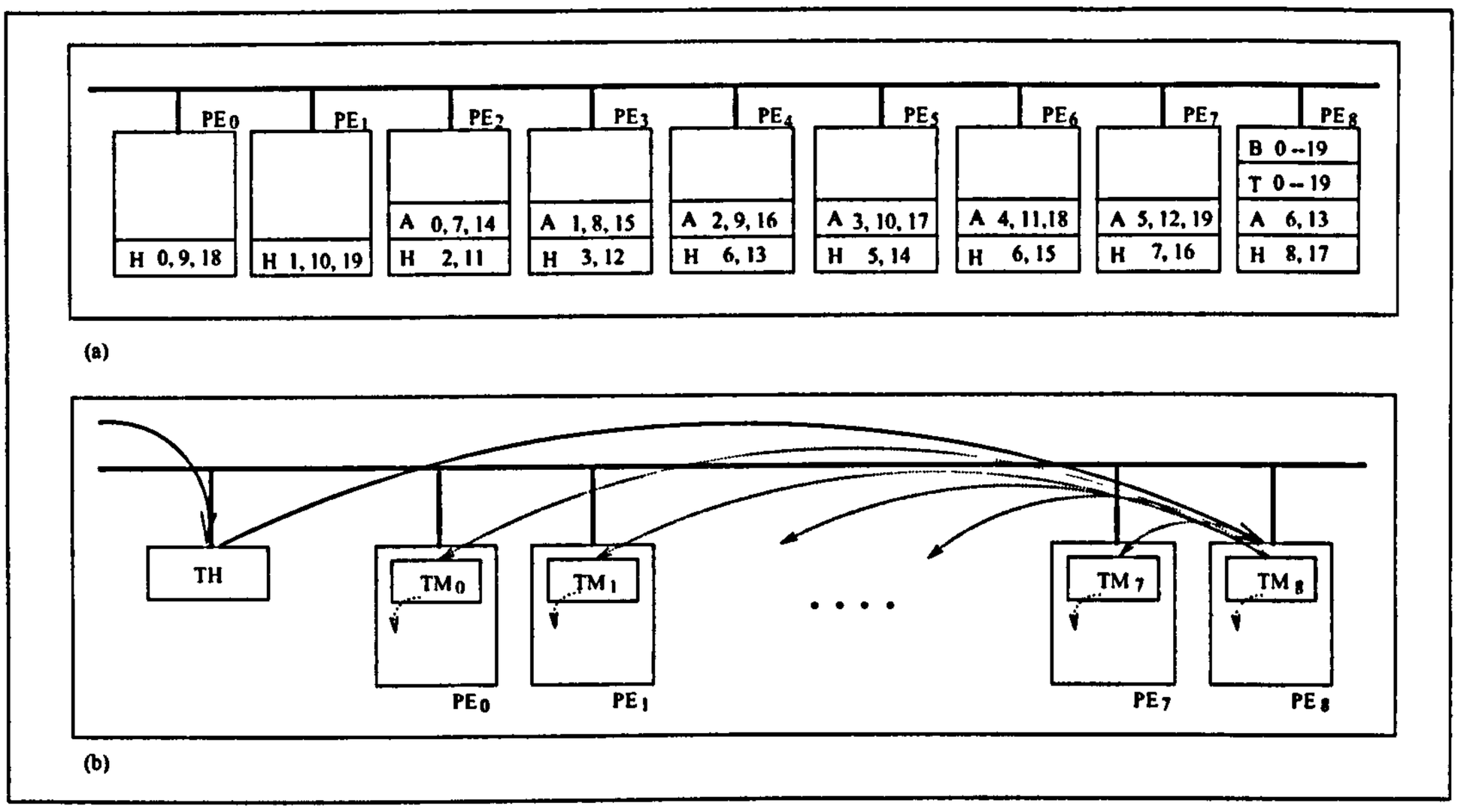


Figure 7.6. (a) Data placement for 9 PE case.
(b) Transaction requests and remote requests distribution.

7.2.4 System Performance Verification

Although the PEPA models may have captured the system behaviour correctly, the ability to accurately estimate the system performance has yet to be proven. It was therefore decided to conduct the same set of experiments on the simplified Ingres Cluster DBMS using an analytical system throughput estimator called STEADY [30, 112]. **Figure 7.7** summarises the results obtained using the PEPA approach and those produced using STEADY. Apart from the case of 7 PEs (which is missing because the state space is too large to be reduced sufficiently to converge on a solution even with the decompositional evaluation approach), both sets of results are in good agreement. This provides a useful validation of PEPA in parallel database system performance modelling.

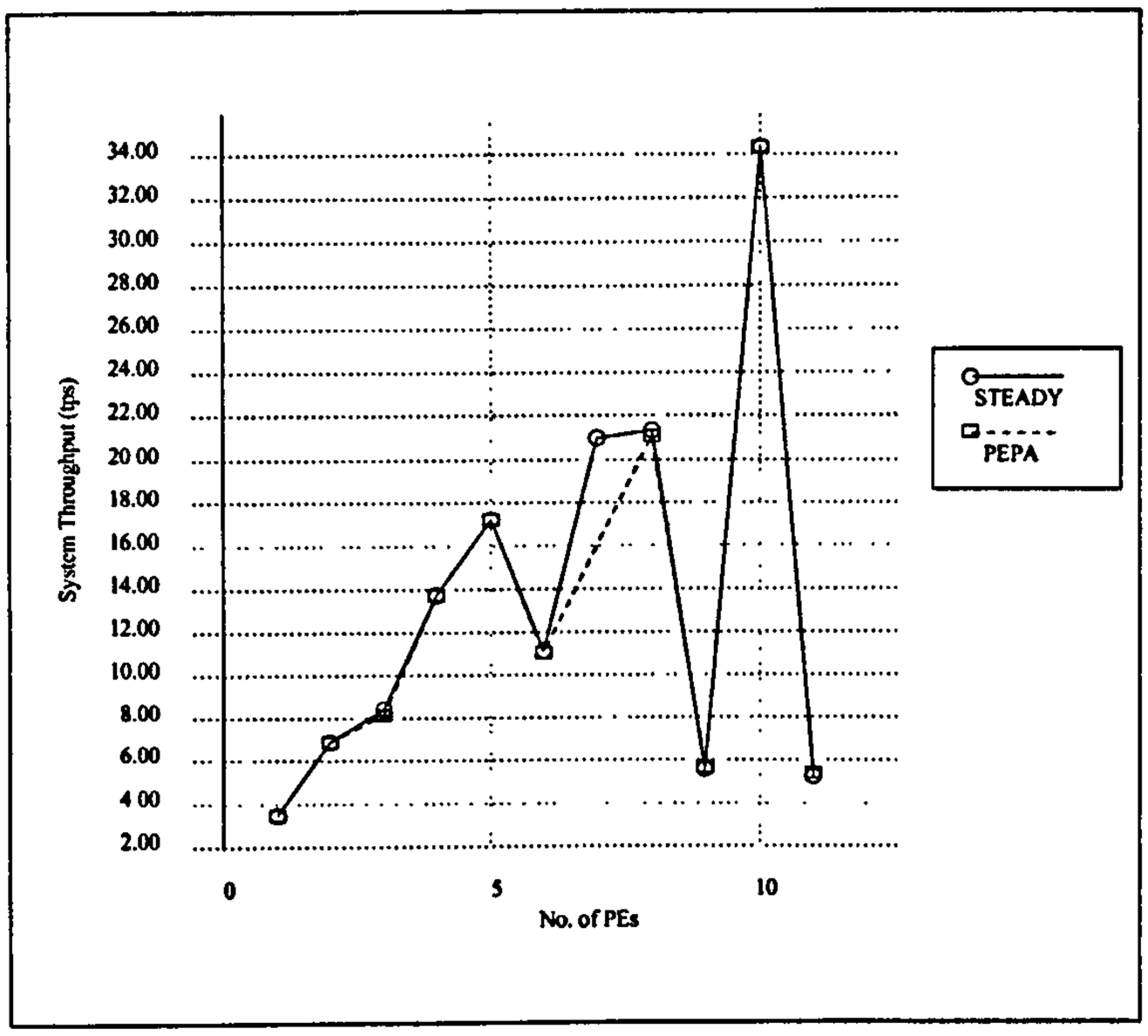


Figure 7.7. The system throughput (tps) for TPC-B estimated by PEPA and STEADY for Ingres Cluster DBMS platform in which the number of PEs varies from 1 to 11.

7.3 Modelling Informix XPS DBMS

Informix XPS DBMS is a commercial product that utilises both inter-transaction and intra-transaction parallelism. In this section, PEPA is used to capture the behaviour of the Informix XPS DBMS running on the ICL Goldrush platform and supporting the TPC-B transaction processing benchmark. The study focuses on the utilisation of both inter-transaction and intra-transaction parallelism of the system. For this purpose, the co-server of the Informix DBMS has been simplified to consist of only one request manager (or transaction manager), one lock manager, one scheduler (or concurrency control unit) and one buffer manager.

It is assumed that there is a co-server whose job is to handle the requests from the users and divide each request into subqueries, and subsequently distribute the subqueries amongst the PEs. This co-server is named transaction host (TH) in this case. The structure of the simplified Informix XPS DBMS is therefore similar to the one shown in **Figure 7.1.**

It is also assumed that each buffer manager has a limited cache that allows several data pages to be kept after being fetched from the disk. This also allows the buffer manager to update a data page in the cache after reading the page from the disk. However, writing back of updated pages is handled by a background process, which will not be modelled here.

7.3.1 Transaction Handling

When the TH receives a request, it divides the request into several subqueries each consisting of an operational request for a particular relation. According to the TPC-B transaction profile, each transaction request consists of three update operations and one insert operation. Consequently, the TH divides the transaction request into four subqueries: three update-requests and one insert-request. The update requests will be distributed to the appropriate PEs for further processing while the insert-request will be delayed until all three update-requests from the same transaction request have completed. It is assumed that the transaction request has an identity number and every subquery carries the transaction ID.

Since the Informix DBMS follows the principle that the data operations are always executed where the data resides, and the relations are well indexed, it is assumed that the TH always distributes a subquery to the PE that contains the data needed for that subquery. After the distribution, the TH waits for the replies from the PEs before inserting a record into relation *History* and committing the transaction. New transaction requests may arrive and will be dealt with in the interim.

Once the update-requests are completed, the TH will receive replies from the PEs. It matches the transaction ID of each reply and decides if all three update-requests from the same transaction request have been completed. If so, the TH will insert a record into relation *History* and send a message to the user.

When the transaction manager on PE_i (TM_i) receives a subquery from the TH, it sends a request to the local concurrency control unit (CCU_i). Because the subquery contains only a single operational request for a particular data item that resides on the PE, the TM_i needs not involve other TMs in the transaction processing. The TM_i waits for the reply from the CCU_i before sending a reply to the TH. In the meanwhile, the TM_i may receive new subqueries from the TH.

The CCU_i and LM_i are fairly straightforward and need no further elaboration. As for the BM_i, when it receives a request from the CCU_i, it will read a data page from the disk and update the page in the buffer before replying to the CCU_i. Occasionally, it just needs to perform the updating in the buffer when the page is found in the buffer.

7.3.2 Representing the System with PEPA

Basically, the PEPA model consists of one component *TH* and several components PE_i, each consisting of several components. The specification of each component PE_i is fairly similar to the one of *ModelA* in *Chapter 5*. Hence, the PEPA model for *ModelA* can be used as the PEPA model for each component PE_i with some modification of components LM_i and BM_i of each component PE_i. Nevertheless, the specification of component LM_i is similar to component LM_i of the simplified Ingres Cluster Model in the previous section.

The specification of component BM_i depends on the data placement on the PE. However, there is only a minor difference between the components BM_i of each PE. Thus, the specification of component BM_i can be expressed as follows,

$$\#BM_i = (ccu_i2bm_i, infty).BM'_i;$$
$$\#BM'_i = (hit, rhit).BM_ih + (miss, rmiss).BM_im;$$
$$\#BM_ih = (update\_buffer, rupbuf)BM_ix;$$
$$\#BM_im = (deliver, rdlv_0).BM_ix + (deliver, rdlv_1).BM_ix + (deliver, rdlv_2).BM_ix;$$
$$\#BM_ix = (bm_i2ccu_i, rs).BM_i;$$

Here, the number of choices of activity *(deliver, $rdlv_i$)* depends on the number of relations allocated to the PE. The time taken to perform the activity *(deliver, $rdlv_i$)* includes the time spent to read a data page from the disk and the update-time of the data page in the buffer. It takes 34.44 msec to read a page from the disk and 825.23 μsec to update a row in the buffer (based on the figures from [30]).

The specification of component TH depends on the data placement that in turn depends on the configuration of the system. Once again, it will be discussed in terms of separate case studies.

7.3.3 Case Studies

A. 3 PE Case

The data placement for this case is similar to the one in **Figure 7.2(a)**. When the TH receives a request from a user, it divides the request into subqueries and subsequently distributes the subqueries to the PEs that contain the data. All three PEs will receive the subqueries from the TH. However, PE_0 and PE_1 will receive more subqueries than PE_2 because of the larger total number of fragments stored on the disk associated with the PEs. After the distribution, the TH waits for the replies from the PEs. Meanwhile, new requests may arrive and will be dealt with.

As mentioned earlier, the relations are fragmented according to a hash function applied to the key attribute Branch-ID of each relation. When a request is divided into subqueries, each subquery carries a transaction ID. It also carries an operational condition which states the value of the key attribute Branch-ID of the target record. For this reason, each subquery is distributed to the PE that contains the relation fragment for which fragment ID matches the value of the key attribute Branch-ID of the condition.

When the TH receives replies from the PEs, it matches the replies according to the transaction ID. However, in this study the TH is assumed to match the replies according to the value of the key attribute Branch-ID because the transaction ID is not explicitly modelled here. For instance, the TH may match all three replies from PE_0 (for fragment ID equal to 8 and 14 for relation *Branch*, *Teller* and *Account*) or PE_1 (for key attribute value 9 and 15) in order to insert a record into relation *History* before committing the transaction. It may also match two replies from PE_0 with one reply from PE_1 (for key attribute value 6, 12 and 18), and so on. Thus, the PEPA specification of component *TH* can be expressed as follows.

$$\#TH^*a = (request, r\_req).TH'a;$$
$$\#TH'a = (reqB, rs).(rmrq, r_0).(reqT, rs).(rmrq, r_0).(reqA, rs).(rmrq, r_0).TH^*a;$$

$$\#TH"a = (rmrq, infty).TH\hat{}a;$$
$$\#TH\hat{}a = (rmreq_0, rq_0).TH"a + (rmreq_1, rq_1).TH"a + (rmreq_2, rq_2).TH"a;$$
$$\#THa = TH^*a \ <rmrq> \ TH"a;$$

$$\#THb = (rmrp_0, infty).TH_0b + (rmrp_1, infty).TH_1b + (rmrq_2, infty).TH_2b;$$
$$\#TH_xb = (write\_his, rs).(reply, r\_reply).THb;$$
$$\#TH_0b = (rmrp_0, infty).TH_{0*}b + (rmrq_1, infty).TH_{01}b + (rmrp_2, infty).TH_{02}b;$$
$$\#TH_{0*}b = (rmrp_0, infty).TH_xb + (rmrp_1, infty).TH_xb + (rmrp_2, infty).TH_xb;$$
$$\#TH_{01}b = (rmrp_0, infty).TH_xb + (rmrp_1, infty).TH_xb;$$
$$\#TH_{02}b = (rmrp_0, infty).TH_xb;$$
$$\#TH_1b = (rmrp_0, infty).TH_{01}b + (rmrp_1, infty).TH_{1*}b + (rmrp_2, infty).TH_{12}b;$$
$$\#TH_{1*}b = (rmrp_0, infty).TH_xb + (rmrp_1, infty).TH_xb + (rmrp_2, infty).TH_xb;$$
$$\#TH_{12}b = (rmrp_1, infty).TH_xb;$$

$$\#TH = THa <> THb;$$

The evaluation of this model follows the decompositional evaluation approach. The model is decomposed into submodels *THa* and *THb*, and several submodels *PE* each of which is further decomposed into submodels (TM_ia, TM_ib, CCU_ia and BCD_i). The submodels are evaluated in turn and the throughput estimated for this case is 25.97 tps.

B. 5 PE Case

The data placement for this case is similar to the one in **Figure 7.5(a)**. As we already know, the TH will receive requests from the users and subsequently divide each request and distribute the subqueries. The point is that when the TH receives replies from the PEs, it matches all three

replies from the same PE_i before inserting a record into relation *History* and committing the transaction. However, the three replies may not arrive in a sequence one after another as there are replies from other PEs that may arrive in the interim and all these possibilities need to be considered when modelling the TH. As a result, the state-space of the submodel *THb* is too large to be reduced sufficiently for a solution. For this reason, intermediate threading is used.

Five intermediate processes are introduced into submodel *THb*, each of which handles the replies from one PE_i and inserts a record into relation History, while component *THb* just has to handle the replies from the intermediate processes without having to worry about the matching of the replies **(Figure 7.8)**. The throughput estimated for this case is 48.51 tps.

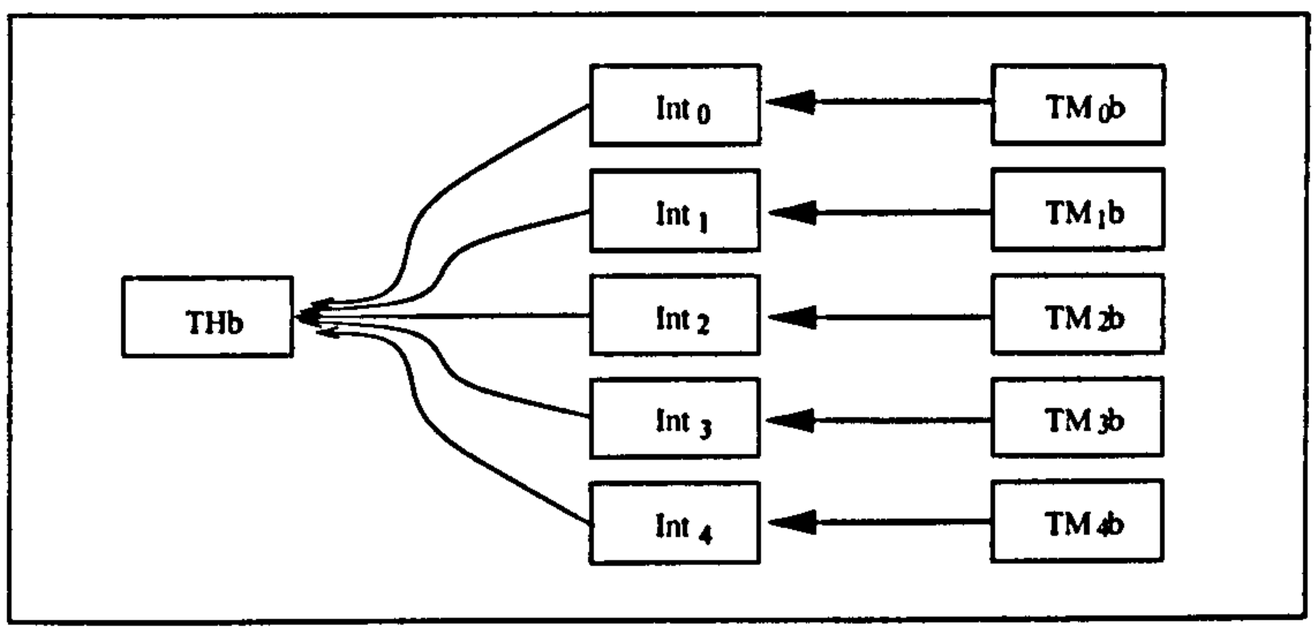


Figure 7.8. Component *THb* and the intermediate processes.

C. 9 PE Case

The data placement for this case is similar to the one in **Figure 7.6(a)**. In this case, when the TH distributes the subqueries amongst the PEs, no subquery is distributed to PE_0 and PE_1. This is because both PE_0 and PE_1 have no fragment of relation *Branch*, *Teller* or *Account* allocated to them. In contrast, PE_8 will receive most of the subqueries as all fragments of relations *Branch* and *Teller* are stored on this PE.

When the TH receives replies from the PEs (except PE_0 and PE_1), it matches two replies from PE_8 with one reply from any of the other PEs. There are only two occasions when the TH

matches all three replies from PE_8 alone (when the key attribute value is 6 and 13). The modelling of this case follows that of the previous cases and the throughput estimated for this case is 14.41 tps.

Figure 7.9 summarises the results for this experiment. The results are compared to those obtained from STEADY. Apart from the missing 7 PE case, both sets of results show fairly good agreement. In the 10 PE case, the arrival rate (100) is similar to the maximum system throughput. As the request arrives randomly, there is an occasion that the system may be busy processing other requests such that the arriving request is blocked waiting in a queue. There may be other occasions when the system is idle such that any arriving request can be processed immediately. For this reason, the throughput estimated using PEPA approach is lesser than the maximum throughput as predicted by STEADY.

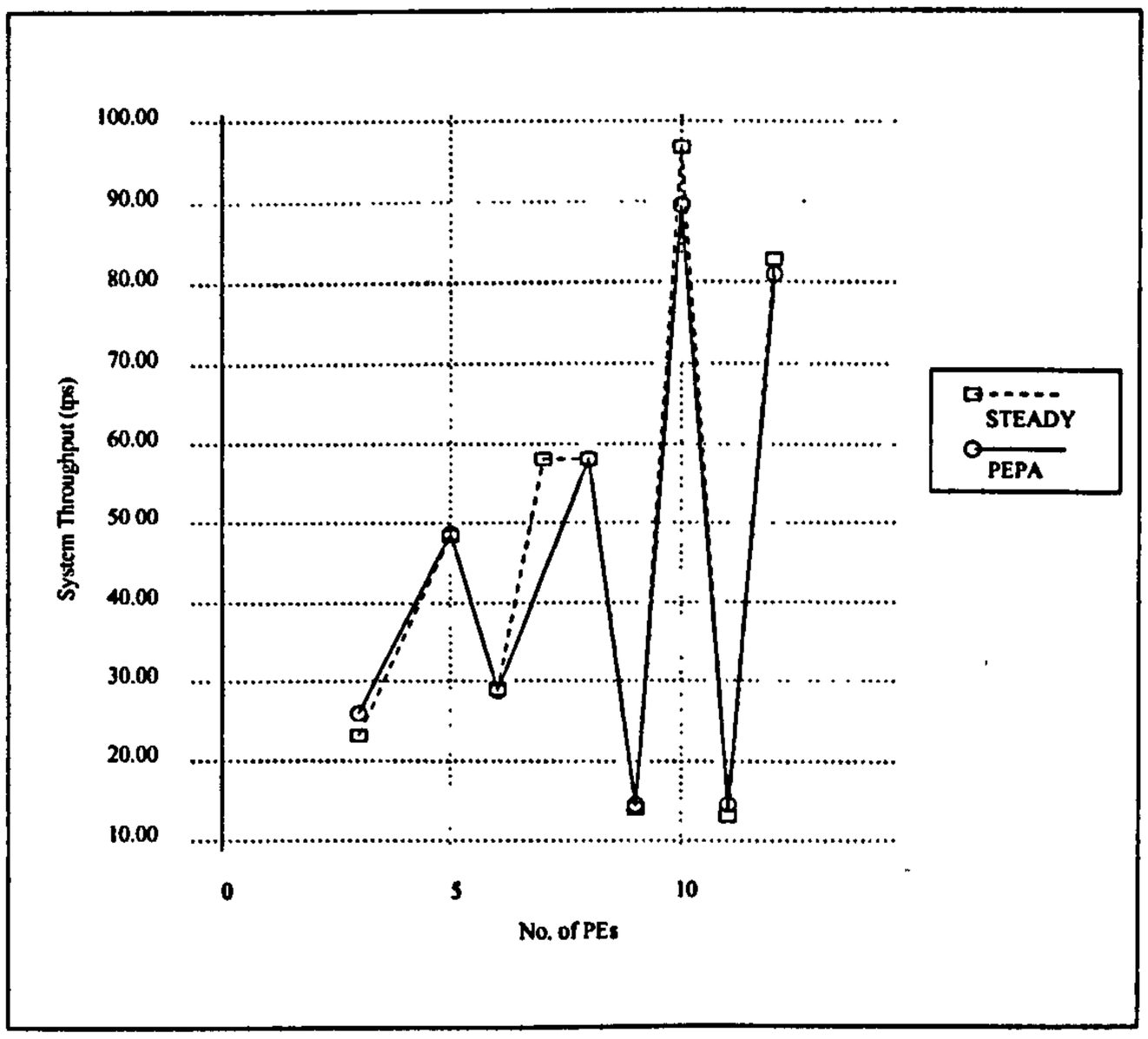


Figure 7.9. System throughput (tps) of PEPA and STEADY for the Informix DBMS when the number of PEs varies between 3 and 12.

7.4 Summary

PEPA has been used to model two actual parallel DBMSs running on a shared nothing Goldrush platform. Both Ingres Cluster DBMS and Informix XPS DBMS have been studied. Despite the simplification of such system in order to suit the purpose of this study, the results obtained using the PEPA approach have been compared to those estimated by STEADY and found to be in fairly good agreement.

Of the two systems, one utilises inter-transaction parallelism while the other utilises both inter-transaction and intra-transaction parallelism. The former one does not divide the request into subqueries but requires the TM_i to send remote requests to other TMs for transaction processing. The handling of the remote replies relies on the communication between these TMs. On the other hand, in the latter system that utilises both forms of transaction parallelism, the requests are divided into subqueries that are subsequently distributed to the PEs that contain the data. For this case, the task of handling replies falls heavily on the transaction host. Nevertheless, the behaviour of both systems depends on the data placement that in turn depends on the system configuration.

In terms of performance, the Informix DBMS model does better compared with the Ingres DBMS model. This is attributed to the fact that the Ingres DBMS model immediately writes back every updated page for every update operation. In contrast, in the Informix DBMS model, the writing back is deferred and is done in the background. Consequently, for every committed transaction, the Ingres DBMS model spends twice as much time doing I/O operations compared to the Informix DBMS model. In addition to this, the cache in the Informix DBMS model allows a small number of data pages to be temporary memory-resident. This also helps to reduce the overall I/O operation times consumed by the Informix DBMS model. All these factors contribute to the difference between the sets of results.

Chapter 8

DATA PLACEMENT IN PARALLEL DBMS

8.0 Introduction

In the previous chapter, we have studied the behaviour of a system that utilises both inter-transaction and intra-transaction parallelism. Given the same dataset and data placement strategy but varying system configurations including the number of PEs in the system, the behaviour as well as the performance of the system changes significantly according to different data distribution patterns. This shows that the data distribution pattern can have a significant effect on the performance of a parallel DBMS especially in the case of shared-nothing platforms. In short a proper data placement is important for load balancing [29, 98].

This chapter takes a closer look into the placement of data on parallel DBMSs. The following section discusses the general idea of data placement. Section 8.2 presents some works related to data placement and its relative effect on the performance of a parallel DBMS. Section 8.3 presents an experiment that focuses on a shared-nothing parallel DBMS (Ingres Cluster DBMS) with multiple disks attached to each node.

8.1 Data Placement Process

The purpose of distributing the database across the PEs is to allow the parallel DBMS to exploit the I/O bandwidth of the multiple disks by reading and writing them in parallel [74]. The data placement process can be divided into three phases.

1. *Partitioning phase* in which each relation is partitioned into fragments.
2. *Distribution phase* in which the fragments of the relations generated in the partitioning phase, are distributed to the PEs in the system.
3. *Reorganisation phase* in which the data is redistributed to restore the load balance of the system after insertions and deletions.

8.1.1 Data Partitioning

In this stage, each relation is partitioned into fragments either horizontally or vertically. Horizontal partitioning partitions a relation at the level of its tuples so that each fragment contains a subset of the tuples of the relation. There are three fundamental strategies for doing this. The simplest one distributes the tuples to the fragments in a *round-robin* fashion. This is the default partitioning strategy in the Gamma database machine. *Hash* partitioning strategy distributes the tuples amongst the fragments according to the hash value obtained by applying a hash function to the key attribute of each tuple. This strategy is implemented in the Bubba, Gamma and Teradata database machines. *Range* partitioning strategy on the other hand clusters tuples with similar attributes together in the same fragment. The user may specify a range of values for the key attributes to be assigned to each fragment. Bubba, Gamma and Tandem provide this partitioning strategy.

Vertical partitioning is more complicated than horizontal partitioning. It partitions a relation so that each fragment contains a subset of the attributes of the relation including the primary key of the relation. This approach is less common in parallel DBMSs.

Apart from these data partitioning strategies, Hua et al [62] have suggested an adaptive data placement scheme for parallel DBMSs. According to their strategy, a relation is partitioned into fragments using multiple attributes of the relation. The partitioning is based on a grid file [85] concept in which the linear scales are used to specify the key ranges in each of the dimensions

(key attributes). Li et al [74] also adapt the grid file concept in their database declustering method.

Ghandeharizadeh et al [44] have proposed a range-based partitioning strategy which is known as *hybrid-ranges* partitioning strategy. According to this strategy, a relation is partitioned into fragments such that each fragment contains a distinct range of the partitioning attribute values. The strategy will determine the number of fragments into which a relation must be fragmented, based on the appropriate degree of intra-query parallelism for the set of queries that access the relation.

8.1.2 Data Distribution

There are many data distribution strategies available. The simplest one is to distribute the fragments to the PEs in a *round-robin* fashion. For instance in [44], the fragments are distributed to the PEs in this way to ensure that adjacent fragments are assigned to different PEs. The fragments can also be distributed to the PEs via a *hash* function such as in Gamma and Teradata. Another common data distribution strategy is the *range* placement strategy where a relation is partitioned according to the key ranges and the fragments are subsequently distributed to the PEs according to the range specified by the user.

The data placement strategy suggested in [62] places the fragments (cells) according to the size of each fragment. It starts the distribution with the largest fragment and allocates this fragment to the PE that currently has the most space available. It continues with the second largest fragment and proceeds in this way until all fragments are distributed across the PEs.

On the other hand, Copeland et al [29] allocate the fragments to the PEs according to the *heat* or access frequencies of the relation. It starts with a relation that has the highest heat and distributes its fragments to PEs that currently have the least accumulated heat. The process is repeated with the relation that has the second highest heat and so on until all relation fragments

are exhausted. Copeland et al [29] also considers the *temperature* of the relation when distributing the fragments across the PE.

8.1.3 Data Reorganisation

The placement of data changes continuously as insert and delete operations take place on each PE until the load balance of the system degrades significantly resulting in a decline of the overall system performance. In this situation, data reorganisation is necessary to resume good system performance. However, there is a cost involved in data reorganisation and Copeland et al [29] has suggested that the data reorganisation should only take place when the benefit outweighs the cost. Should data reorganisation take place, the data fragments can be reshuffled according to their initial placement strategy or a different placement strategy.

Nevertheless, Hua et al [62] have also proposed a data reorganisation algorithm that may restore the load balance of the system at as little cost as possible. The data fragments on each PE are sorted into sorted lists on each PE. Each PE will first retain the largest fragment from its sorted list. The PE with the largest total size of fragments will determine the total size of fragments which other PEs have to achieve by adding more fragments from their own sorted lists. The process is repeated until one of the PEs runs out of fragments in its own sorted list. The remaining fragments on other sorted lists will be sorted in a single sorted list on a host PE before these fragments are re-distributed to the PEs using the initial distribution strategy.

Chamberlin et al [20] have presented an approach to reorganisation of data to balance the load in a system without interrupting the service. This approach is called *Dynamic Data Distribution* (D^3). According to this approach, each node has a *partition function* (PF) which maps the fragment-ID (block-ID) to its storage site. The PF can be used to locate a block, or select a site for a new block. A node is designated the coordinator that gathers the load statistic and information about other nodes. The coordinator will decide when a new PF to be created to

balance the load of the system. When a new PF is created, it is transmitted to all the other nodes. When a node receives a new PF, it starts scanning through its blocks and applies the new PF to the blocks to determine the new storage sites for them, until all the blocks are stored according to the new PF.

8.2 Data Placement and System Performance

Although various data placement strategies have been developed, little work has been done on assessing the relative performance obtained from different strategies under different circumstances.

Hanson et al [52] have performed a study on the impact of data placement on the performance of a relational database system running on a multiprocessor platform. Both value-range and round-robin partitioning strategies were used. The experiment was tested using a single relational database, which varies from 1 to 50000 records. Their study showed that the value-range partitioning strategy provides a better performance compared to the round-robin partitioning strategy.

Ghandeharizadeh et al [43] have conducted a study on the impact of alternative partitioning strategy and storage organisation on different selection query types on the Gamma database machine. The data placement strategies include *round-robin*, *hash* and *range* while the storage structures selected are *heap*, *clustered index* and *non-clustered index*. The benchmark for the study is based on the Wisconsin benchmark. Their conclusion shows that no placement strategy is superior under all circumstances. Rather, each placement strategy outperforms the others for certain query types.

Zhou et al [110] have conducted a series of studies on various data placement strategies and have compared the relative performance of a shared-nothing parallel DBMS obtained from these

strategies. The study has been conducted using an analytical tool (STEADY). Among the placement strategies studied are Hua's, Bubba's, *size* (which is derived from Hua's algorithm), *heat* (which is derived from Bubba's algorithm) and *size_and_heat*. TPC-B and TPC-C transaction processing benchmarks have been used to test the system performance. Again, each placement strategy outperforms others under different circumstances.

Hanson's investigation that is based on a single relational database experiment may be too simple. Both Ghandeharizadeh and Zhou's studies have confined their attention to the PE level data placement. Ghandeharizadeh et al [43] assume that the fragments are equally distributed amongst the disks while Zhou et al [110] assume that the fragments are distributed to the disks in an *abstract round-robin* fashion. Nevertheless, both studies have shown the significant effect which a data placement strategy can have on overall system performance.

8.3 Dual-level Data Placement

Following the work done by Zhou et al [110], it was decided to extend the work to study the effect that may be observed if the data placement strategies are applied to the disk level. The system under study is Ingres Cluster DBMS running on Goldrush shared nothing platform. Of the range of data placement strategies available, three have been chosen for this study as they typify a number of placement strategies. They are *size*, *heat* and *size_and_heat*, while the data partitioning follows the hash partitioning algorithm. Data reorganisation is not considered. The study is conducted using STEADY.

The *size* placement strategy has been derived from Hua's placement algorithm adapted for the particular system under study by Zhou et al [110]. However, the *size* placement strategy used in this study is slightly different from the one used in [110]. In this study, the *size* placement strategy starts with the smallest relation in the database. The relation is partitioned into fragments which are subsequently distributed among the PEs using a greedy algorithm, that is when a

fragment is distributed, it is allocated to the PE which has the most space available. If more than one PE has the same maximum space available, a PE is chosen arbitrarily. This process is repeated for the second smallest relation in the database and is continued until all relations have been dealt with. The fragments of the relations assigned to each PE are put into a fragment list of the PE.

The approach used for distributing fragments at the disk level is slightly different from the one used at the PE level. At the disk level, provided that all disks start with the same amount of free space, the fragment list associated with the PE is sorted into ascending order of fragment size. The distribution of fragments among the disks associated with a particular PE starts with the smallest fragment in the list. The fragment will be allocated to a disk that has the most space available. This process is repeated for the second fragment from the fragment list and is continued until the list becomes empty. The same process is carried out for every PE.

The *heat* data placement strategy has been derived from [29] approach adapted for the system under study. It uses a placement heuristic based on access frequencies of each relation. The heat of each relation is estimated by analysing the effects of all queries on the relations. At the PE level, the *heat* strategy starts by partitioning the relation with the highest heat value. The fragments of this relation are distributed across the PEs again using a greedy algorithm. Here, a fragment is allocated to a PE for which the accumulated heat values of all fragments assigned to it so far is the lowest among the PEs. The distribution process is repeated until the current relation is exhausted. The partitioning and distribution processes are repeated for the relation with the second highest heat value and so on until every relation in the database has been distributed.

At the disk level, the list of fragments associated with each PE is sorted in descending order according to the heat value. A fragment is allocated to a disk for which the accumulated heat value of the fragments assigned to it so far is the lowest for that PE. This process is repeated for the second fragment and so on until the fragment list is empty. The same process is carried out for every PE.

The *size_and_heat* placement strategy is quite similar to the approach implemented in the other two placement strategies although more complicated than either of them. The difference lies in the weight used to decide the priority of relations to be distributed and to order the fragments as well as to select the particular PE or disk for placement. The weight used for this strategy when distributing fragments across PEs is the product of the relation size and the relation access frequency. When distributing fragments across the disks associated with a PE, the weight used is the product of the fragment size and the fragment access frequency.

8.3.1 The Transaction Benchmark

TPC-C transaction processing benchmark is used to test the system performance. In this study, the metric is the maximum system throughput and the queries are the mixed workload of *order entries*, *customer payment*, *order status check* and *delivery*. All possible combinations of three placement strategies are explored in the context of two main variable factors: number of PEs and database size. A data placement strategy will be selected for the PE level while another (probably the same) placement strategy will be used at the disk level. For instance, a combination *size/heat* means that the *size* placement strategy is used to distribute the data at the PE level while the *heat* placement strategy is used at the disk level.

8.3.2 Experiment 1 - Number of PEs as Variable Factor

Consider the effect of varying the number of PEs from 3 to 59. The size of the database determined by the number of the warehouses is fixed at 84 warehouses and the number of the disks attached to each PE is set to be two, four and six. **Table 8.1** shows the details of the

database while **Table 8.2** displays the relative access frequencies of the mixed workload for the TPC-C benchmark.

Relation Name	No. of Attributes	Tuple size (bytes)	No. of Tuples	Total size (bytes)
Warehouse	5	100	84	8400
District	7	100	840	84000
New_order	4	20	8400	168000
Item	7	100	6000	600000
Parts	3	50	13200	660000
Stock	3	20	84000	1680000
Order	7	50	840000	42000000
Customer	18	400	840000	336000000
History	6	50	7560000	378000000
Order_line	10	50	8400000	420000000

Table 8.1. TPC-C database.

Query	Frequency
Order Entries	48%
Customer Payment	48%
Order Status Check	2%
Delivery	2%

Table 8.2. Query frequencies in TPC-C mixed workload.

Figure 8.1 summarises the performance measured in terms of maximum system throughput obtained from the data distribution generated by each combination of placement strategies for the TPC-C database, when the number of disks attached to each PE is two. In each case, the disks attached to particular PEs are the bottlenecks.

The TPC-C database has several large relations (*Order_line*, *History* and *Customer*) whose sizes are similar. Of these three relations, *Order_line* and *Customer* are accessed more frequently in the mixed workload of the TPC-C benchmark than other TPC-C relations. The number of read/write operations required for small relations such as *Warehouse* and *District* is relatively very small in this workload. As a result, the placement of the fragments of *Order_line* and *Customer* has the greatest effect on the system performance. Because of the high access frequency of *Order_line* and *Customer* relations, the disks with the greatest number of

Order_line and *Customer* fragments assigned to them will become the system bottlenecks and thus determine the overall system throughput value.

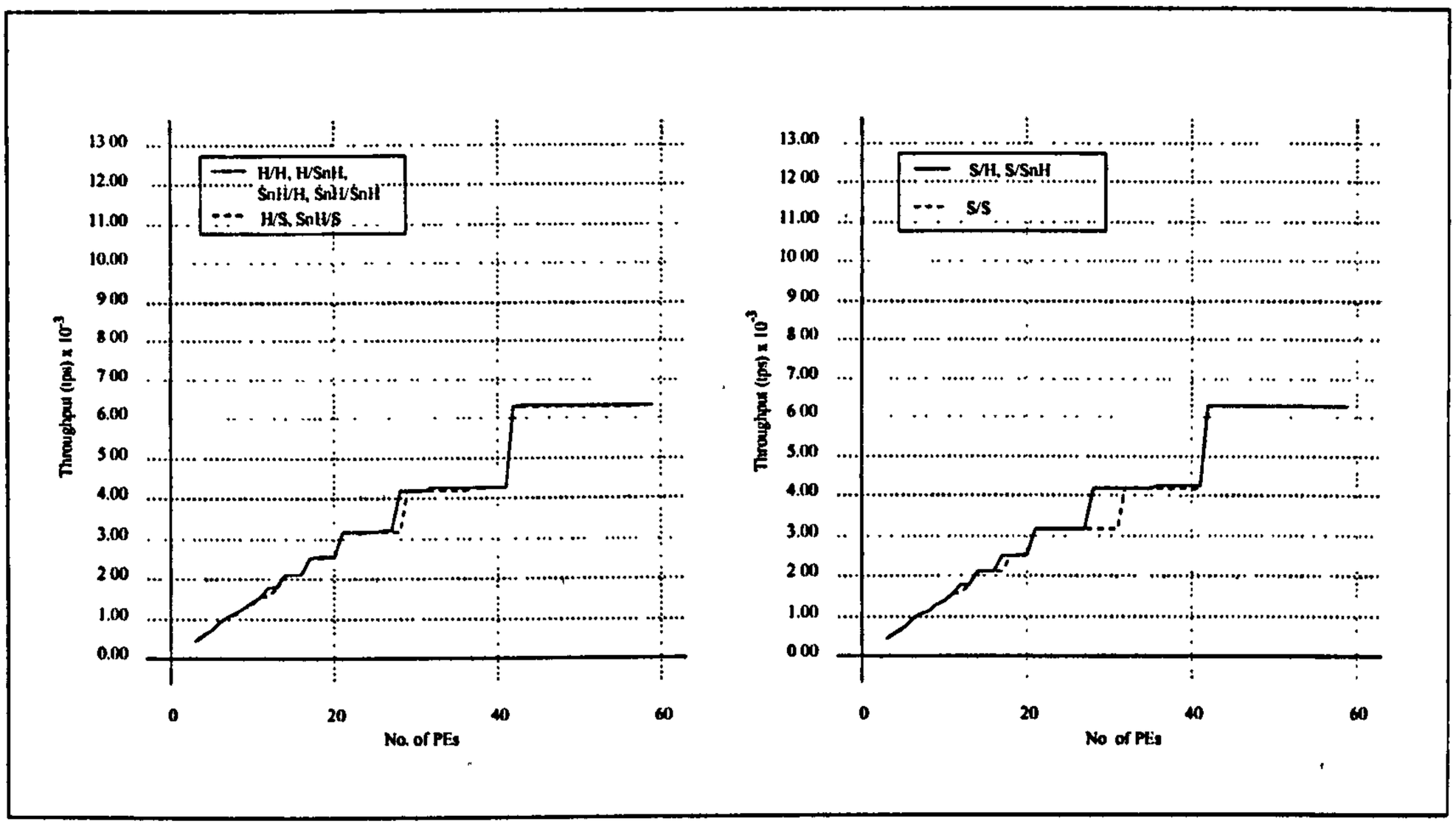


Figure 8.1. System throughputs (tps) obtained using different combinations of data placement strategies with varying number of PEs, 2 disks per PE and 84 warehouses.

Although nine possible combinations have been investigated, several of these combinations produce identical or almost identical results. In particular, both *size_and_heat* and *heat* placement strategies place *Order_line* and *Customer* fragments first and subsequently generate identical distribution pattern for both relations and hence produce similar system performance behaviour. Consequently, only four distinct graphs are shown.

Each of the graphs displays step-like performance behaviour. When the number of PEs is small (between 3 and 14), changes in the number of PEs has a significant effect on the system performance. This is because when the number of PEs is small, each additional PE reduces the total number of *Order_line* and *Customer* fragments allocated to disks and hence reduces the access frequency to individual disks. As the number of PEs increases (above 14), additional PEs do not always change the total number of *Order_line* and *Customer* fragments on the bottleneck disks but may reduce the number of bottleneck disks.

There are no significant variations between the performance curves except at the points when the placement algorithms produce another level of system performance. Each performance curve has an increment when the number of PEs is 14, 21 and 42, i.e. when the number of fragments is an exact multiple of the number of disks in the system. For these cases, all placement algorithms produce the same optimum distribution. Performance increments also occur at other points that vary for different placement algorithms. For instance, the combinations of the *heat* and *size_and_heat* placement strategies as well as the *size/heat* and *size/size_and_heat* placement strategy combinations produce changes at 17 PEs and 28 PEs. Whereas the combinations of *size/size* produces changes at 18 PEs and 32 PEs, and *heat/size* at 17 PEs and 29 PEs. The changes in the performance curves for these cases are caused by changes in the total number of *Order_line* and *Customer* fragments on the bottleneck disks. As an example, at 41 PEs (for *heat/heat* placement combination), the bottleneck disks each have one *Order_line* and two *Customer* fragments. When the number of PEs increases to 42, the bottleneck disks are each assigned one *Order_line* and one *Customer* fragment. The access frequencies to the bottleneck disks have been reduced and this increases the system performance.

At certain points, the performance curves show relatively small increases. The reason for this is that instead of reducing the total number of *Order_line* and *Customer* fragments on the bottleneck disks, one of the *Order_line* fragments is swapped with a *Customer* fragment. For example, using the *heat/heat* placement combination with 32 PEs, each bottleneck disk is allocated two *Customer* and one *Order_line* fragments. By comparison, for the case of 31 PEs, the bottleneck disks are assigned with two *Order_line* and one *Customer* fragments each. Since the access frequency of the relation *Order_line* is higher than that for the relation *Customer*, reducing the number of *Order_line* fragments will reduce the number of accesses to the bottleneck disks and hence yield a better system throughput.

The combinations of *heat* and *size_and_heat* placement strategies produce the best average system performance among the placement algorithms. However, there is no significant

difference in the results produced by different combinations for the case when two disks are attached to a PE (with less than 3% variation in the average system throughput **(Appendix G)**).

In the case of four disks per PE **(Figure 8.2)**, five distinct performance curves are shown. The performance curves display similar performance behaviour to the previous case. However, the steps are shorter and the throughput higher. All performance curve have increments at 3 PEs, 7 PEs and 21 PEs (which correspond to situations when the number of fragments is an exact multiple of the number of disks and all placement algorithms produce the same optimum distribution). Apart from these, all placement combinations also show increments in performance at other points. For instance, the combinations of the *heat* and *size_and_heat* placement strategies also display increments in performance at 9 PEs, 11 PEs, 14 PEs, 16 PEs, 32 PEs and 42 PEs.

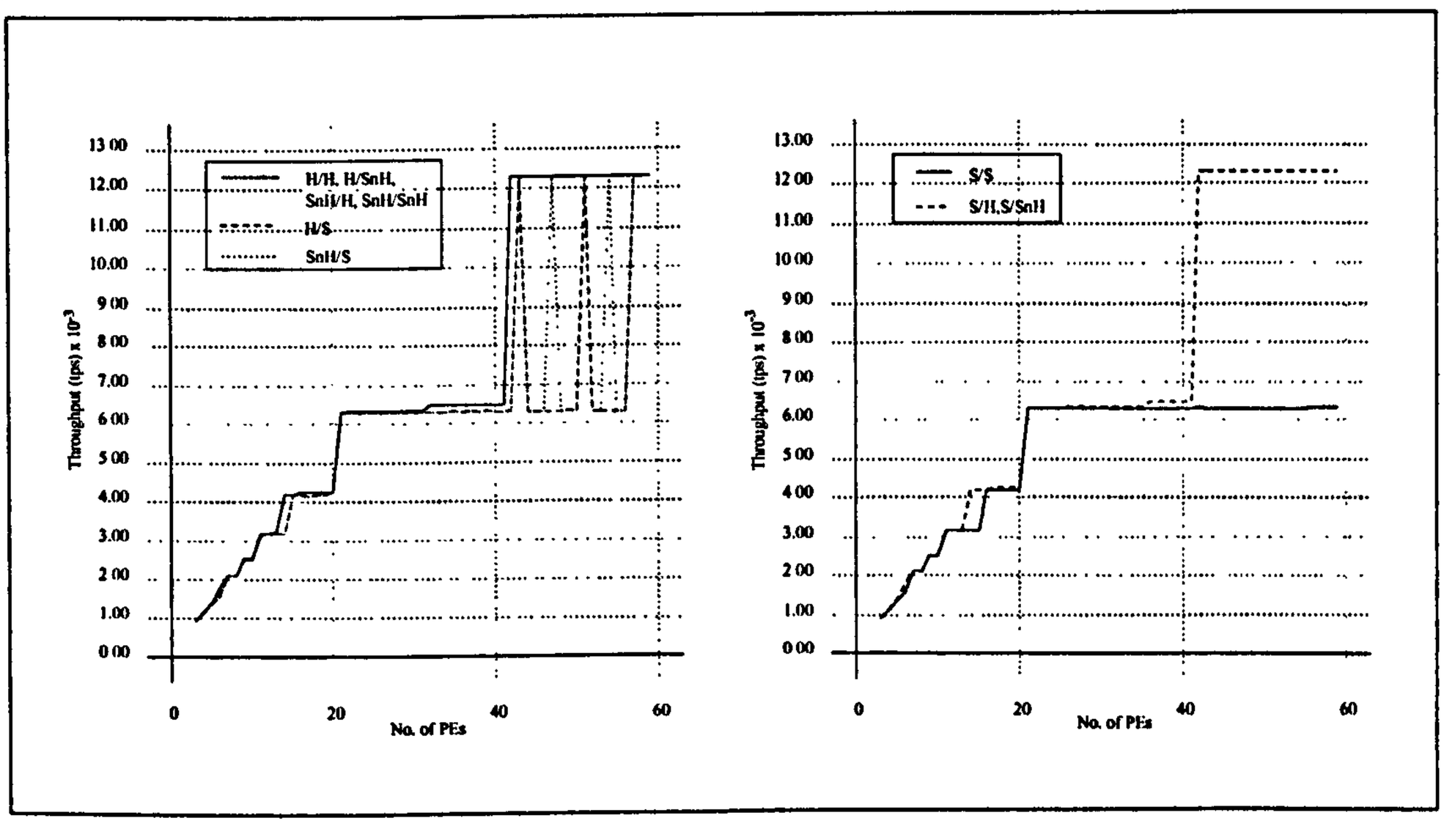


Figure 8.2. System throughputs (tps) obtained using different combinations of data placement strategies with varying number of PEs, 4 disks per PE and 84 warehouses.

At 42 PEs and above, the system has sufficient disks to allocate the *Order_line* and *Customer* fragments to different disks. In the TPC-C benchmark, the fragment of *Order_line* is the finest granularity that determines the system performance. The combinations of *heat* and *size_and_heat* placement strategies, and also *size/heat* and *size/size_and_heat* placement

algorithms produce the optimum system throughput for these cases. However, the remaining combinations of placement algorithms do not produce the optimum system throughput in every case above 42 PEs.

The maximum system throughput generated by the *size/size* placement strategy combination is a maximum from 21 PEs to 59 PEs. This is because the placement algorithm has allocated one *Order_line* and one *Customer* fragment to the bottleneck disks for these cases.

For the *heat/size* and *size_and_heat/size* placement combinations, the performance curves are rather unstable above 42 PEs. The fragments of all relations distributed at the PE level are based on the relation's heat values or the product of the heat values and sizes. Therefore, most of the fragments that are relatively less frequently accessed may have been distributed to particular PEs instead of being distributed in a uniform way across all PEs. When these fragments are allocated to disks using *size* placement strategy, PEs with an excess number of fragments of relations such as *History* will cause the *Customer* fragments to be allocated to the same disks as the *Order_line* fragments are assigned to.

If six disks are attached to each PE (**Figure 8.3**), the step-like behaviour is again observed. All performance curves show common increments in performance at 7 PEs and 14 PEs when the number of fragments is an exact multiple of disks. Performance curves also show increments at other numbers of PEs but these vary for different placement algorithms.

The combinations of the *heat* and *size_and_heat* placement strategies, as well as the *size/heat* and *size/size_and_heat* placement strategy combinations, produce the optimum system throughput when the number of PEs is 28 and above. This is because the system has sufficient disks to allocate the fragments of the two most frequently accessed relations (*Order_line* and *Customer*) to different disks.

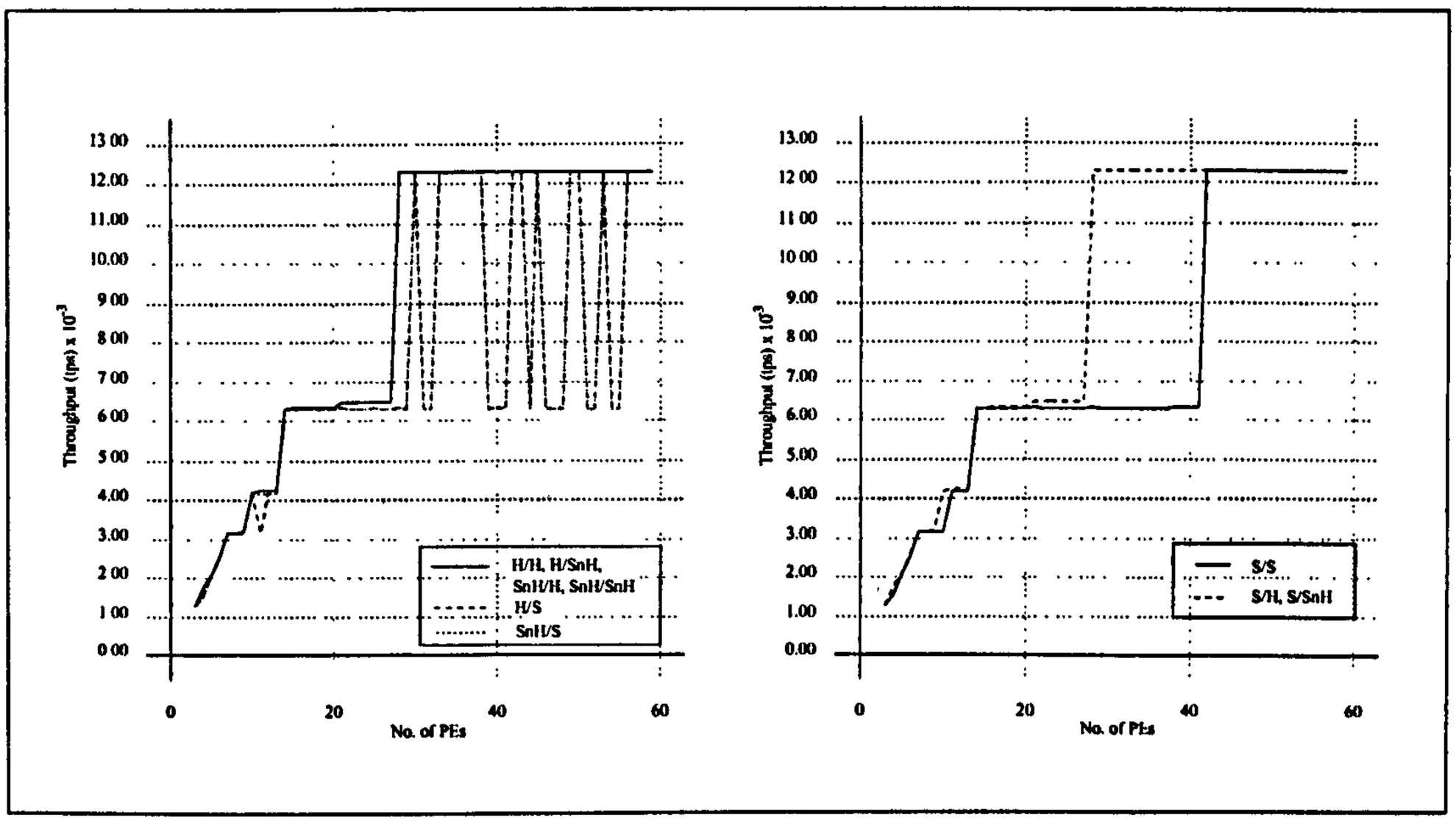


Figure 8.3. System throughputs (tps) obtained using different combinations of data placement strategies with varying number of PEs, 6 disks per PE and 84 warehouses.

The *size/size* placement algorithm only produces the optimum system performance at 42 PEs and above when the number of disks is sufficient to allocate each fragment of the three biggest relations (*Order_line*, *History* and *Customer*) to different disks. The *heat/size* and *size_and_heat/size* placement strategy combinations are again unable to maintain the optimum system performance for cases above 28 PEs because of the allocation of *Order_line* and *Customer* fragments to the same bottleneck disks as before.

In summary, as the number of PEs varies from 3 to 59, the performance produced by different placement strategy combinations varies. When the number of warehouses is a multiple of the number of disks in the system, all placement strategy combinations studied produce the same increment value. However, when the number of warehouses is not a multiple of the number of disks, significant differences in performance are produced, with the combinations of the *heat* and *size_and_heat* placement strategies producing the best average system throughput for all cases. The *size/size* placement combination generates the worst average system throughput when the number of disks attached to each PE is two and four, while the *heat/size* placement combination produces the worst average system throughput for the case of six disks per PE.

8.3.3 Experiment 2 - Database Size as Variable Factor

This section considers the effects on performance produced by changes to the database size. The number of warehouses in the TPC-C database is varied from 10 to 100, and the number of PEs is kept constant at 14 (a typical number of PEs – a 16 PE system with two dedicated PEs). Again, the number of disks attached to each PE is set to two, four or six. Assume that the number of warehouses is *n*. **Table 8.3** shows the relative size of each relation in the TPC-C database used in this study.

Relation Name	No. of Attributes	Tuple size (bytes)	No. of Tuples	Total size (bytes)
warehouse	5	100	*n*	100*n*
district	7	100	10*n*	1000*n*
new_order	4	20	100*n*	2000*n*
item	7	100	71.43*n*	7143*n*
parts	3	50	157.14*n*	7857*n*
stock	3	20	1000*n*	20000*n*
order	7	50	10000*n*	500000*n*
customer	18	400	10000*n*	4000000*n*
history	6	50	90000*n*	4500000*n*
order_line	10	50	100000*n*	5000000*n*

Table 8.3. The relative size of TPC-C database relations (*n* = number of warehouses).

Figure 8.4 displays the variation in performance in terms of system throughput. The values are obtained from the data distribution generated by each pair of placement strategy combinations for the TPC-C benchmark as the database size varies from 10 to 100 warehouses, while the number of PEs is held constant at 14 and the number of disks attached to each PE is two. In each case, the disks attached to particular PEs are the bottlenecks.

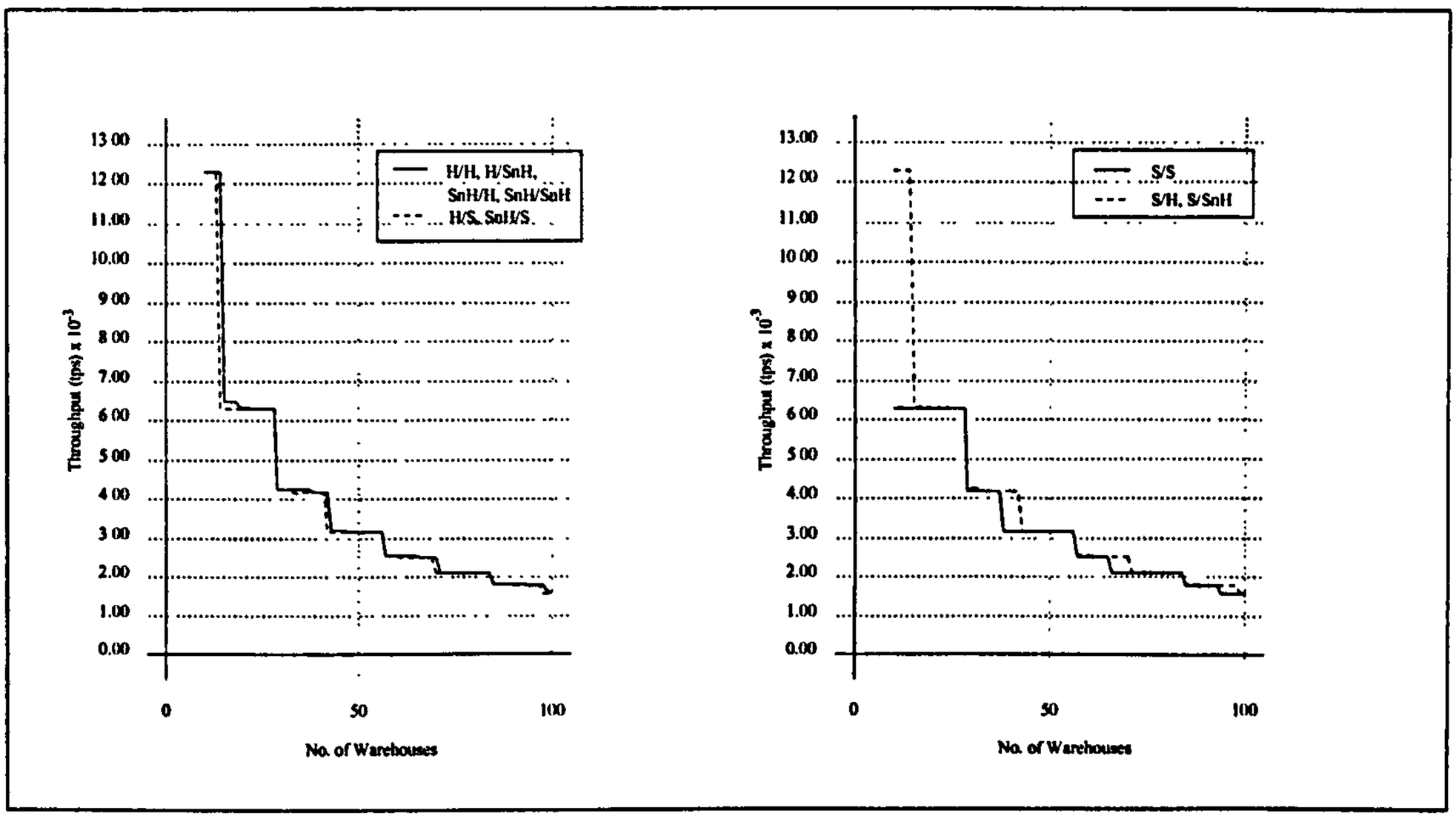


Figure 8.4. System throughputs (tps) obtained using different combinations of data placement strategies with varying database size, 2 disks per PE and 14 PEs.

All placement algorithms produce step-like performance behaviour with common drops in performance when the number of warehouses *n* exceeds 28, 56 and 84 (i.e. when the number of fragments is a multiple of the number of disks and the placement algorithms produce the same optimum distribution). The reason for this is that when the database size increase above these values, the total number of *Order_line* and *Customer* fragments on the bottleneck disks increases by 1. When the number of warehouses is not an exact multiple of the number of disks, the drops in performance occur at different values of *n* for different placement algorithms (for the similar reason explained in the earlier experiment).

Most of the placement combinations produce maximum system throughput above 0.012 tps except the *size/size* placement strategy combination (which is about half of this value). The reason for this is as before (i.e. the bottleneck disks are allocated one *Order_line* and one *Customer* fragment). The average system throughputs produced by the placement combinations are quite similar except for the one produced by the *size/size* placement combination.

For the case of four disks per PE (**Figure 8.5**), all placement combinations produce step-like performance behaviour as the case before with a common performance drop when the number of

warehouses exceeds 56 for the same reason as before. The performance curves also have other performance drops at different numbers of warehouses, according to the placement algorithm used.

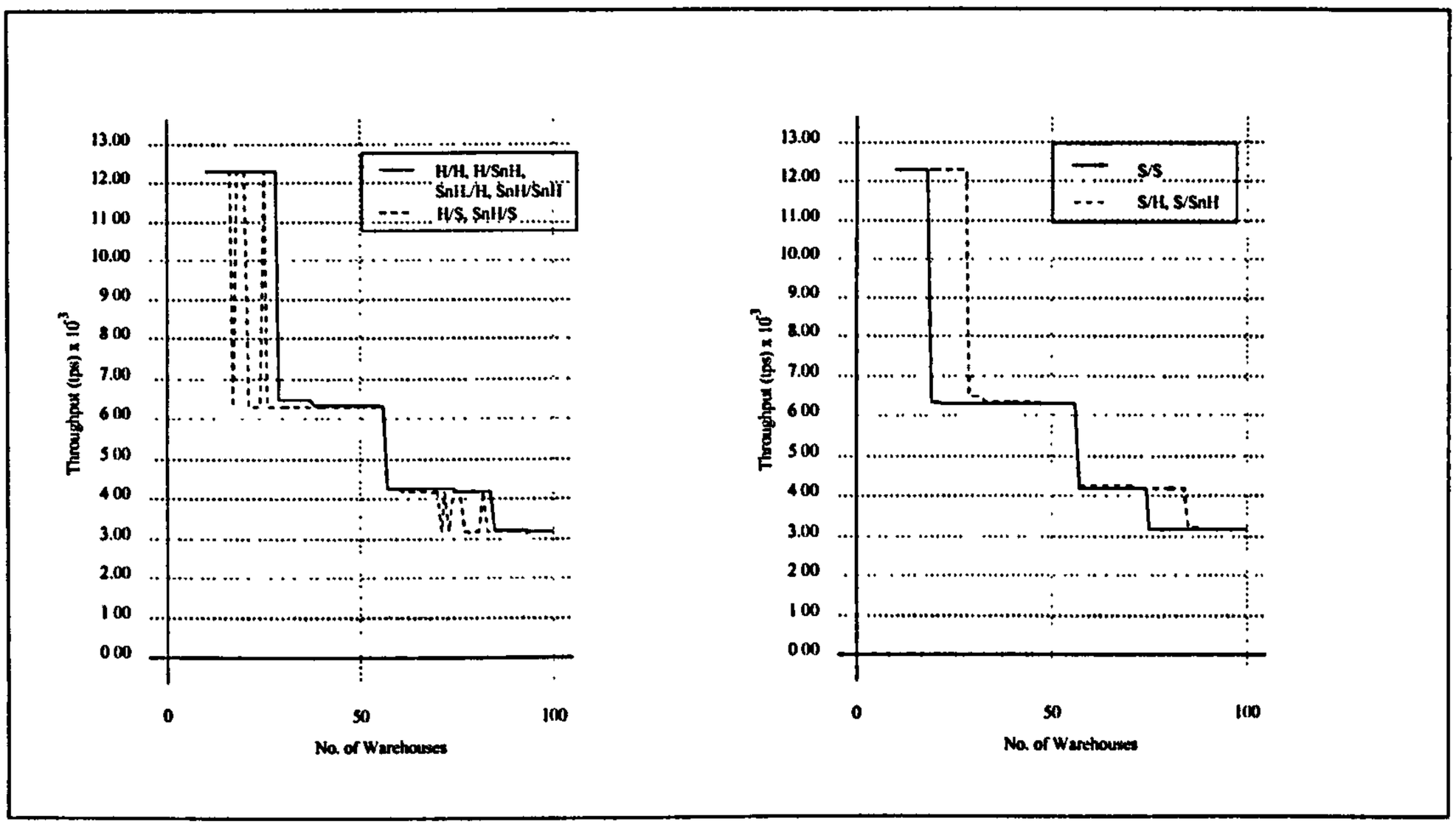


Figure 8.5. System throughputs (tps) obtained using different combinations of data placement strategies with varying database size, 4 disks per PE and 14 PEs.

The *heat/size* and *size_and_heat/size* placement combinations display a rather unstable performance with several drops in the performance before becoming level. As before, some PEs may have been allocated higher numbers of fragments of the relations that have relatively low access frequencies but with larger sizes (e.g. *History*). This disturbs the total number of *Order_line* and *Customer* fragments on the bottleneck disks.

When there are six disks attached to each PE (**Figure 8.6**), the performance drops after the common point when the number of warehouses is 84. This is because of the fact that the number of fragments of each relation is equal to the number disks at that point and an increment in the database size disturbs the balance of *Order_line* and *Customer* fragments. Performance curves also drop at varying points for different placement algorithms. The *heat/size* and *size_and_heat/size* again display variable performance behaviour for the case when the number of warehouses is less than 42 for the same reason as before.

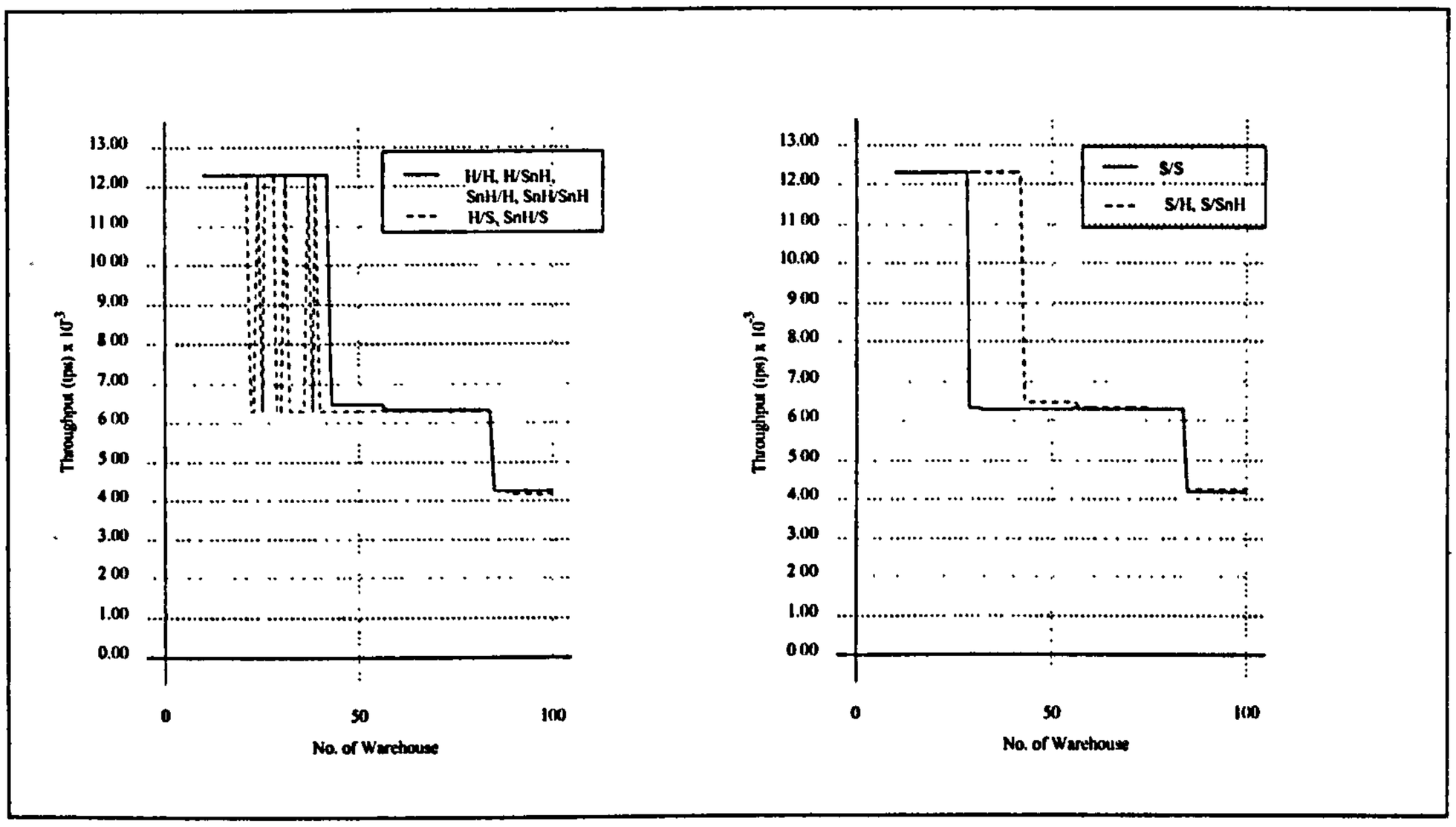


Figure 8.6. System throughputs (tps) obtained using different combinations of data placement strategies with varying database size, 6 disks per PE and 14 PEs.

In summary, all placement strategy combinations produce performance drops when the number of warehouses exceeds a multiple of the total number of disks in the system because of the increment in the total number of *Order_line* and *Customer* fragments. In addition, performance also drops at different numbers of warehouses for different placement algorithms. As the number of warehouses increases from 10 to 100, the variation in performance produced by different strategies becomes less.

8.4 Summary

This study has investigated the effect of using different data placement strategies for distributing data across the PEs and the disks attached to each processing element in a shared-nothing parallel database architecture. Three data placement strategies have been used in different combinations to obtain the best possible placement for the TPC-C transaction processing benchmark under conditions in which the number of PEs has been varied from 3 to 59, and the size of the database

has increased from 10 to 100 warehouses. At the same time, the number of disks attached to each PE has been configured to two, four and six.

Most significant conclusion is that as the number of disks increases, the variation between different data placement strategy combinations increases noticeably. In additional, the results of the studies show that all placement strategy combinations produce changes in performance around some common points when the number of disks in the system is a common factor of the number of warehouses. In the cases when the number of *Order_line* and *Customer* fragments on the bottleneck disks is the same, even if the number of bottleneck disks varies, the throughput is the same.

On average, the combinations of *heat* and *size_and_heat* placement strategies produce the best average system throughput of the 9 placement combinations. This shows that for the TPC-C transaction processing benchmark, the performance of a parallel DBMS is determined to a large extent by the access frequencies. Hence, it can be concluded that in the TPC-C database, the system performance is determined by the placement of the two most frequently accessed relations (i.e. *Order_line* and *Customer*).

Chapter 9

CONCLUSIONS

9.0 Summary

This thesis has demonstrated how a mathematical formalism known as PEPA, a stochastic extension of classical process algebra, can be used to model the performance of a parallel database system. The investigation starts with a simple database system running on a simple platform which has one processing element with one disk attached to it. The system supports a simple query that accesses a simple database.

A PEPA model is constructed for this simple system. The model consists of four components namely TM, CCU, LM and BM. The model is subsequently evaluated and results are obtained. In the experiment, two variable factors are considered: database size and transaction arrival rate. Because PEPA does not actually model the database size, this is reflected in an activity rate of activity accesses to the database. When the database size increases, the activity rate becomes slower.

The main drawback of PEPA is the state space explosion problem. As a model gets larger, the size of the state space grows rapidly until it becomes too large to be solved. Although PEPA does introduce the notions of equivalence as the model simplification technique that can reduce the size of the underlying stochastic process significantly, sometimes the state space is too large to be reduced sufficiently to be able to converge on a solution. For this reason, the decompositional evaluation approach is introduced in *section 5.2.2*.

The decompositional evaluation approach is derived from the flow equivalent aggregation technique of queueing theory. According to this approach, a model is decomposed into submodels, each consisting of one or more atomic components. A queue generator that simulates

the arrival of requests is introduced into every submodel except the one that receives requests directly from the external environment (users). The evaluation follows an iterative approach starting with the submodel that receives the requests from the users. The throughput of this submodel becomes the arrival rate of requests for the queue generator of a subsequent submodel.

Every submodel is evaluated in turn. The throughput of the last submodel evaluated, which sends the replies to the users, becomes the overall throughput of the system. In order to verify the results obtained using the decompositional evaluation approach, the simple model has also been evaluated using the compositional strong equivalence aggregation technique. It is shown in *section 5.2.3* **(Figure 5.3)** that all three approaches, decompositional, strong equivalence and the full PEPA model, have little difference in the performance predicted. This is further confirmed in *section 5.3.1* when the model is extended to handle two disks on a single PE. Thereafter, all the PEPA models studied in this thesis are evaluated using the decompositional evaluation approach.

In *chapter 6*, PEPA is used to model multiple nodes on a parallel database system. Most of the homogeneous PE models reuse the PEPA description for the simple example. In this chapter, the intermediate threading technique is presented. Sometimes, a submodel of a decomposed model may become too large to be resolved. It is therefore important to find a solution to solve the evaluation problem. The intermediate threading technique requires one to identify the underlying parallelism within the submodel and subsequently replace this underlying parallelism with intermediate processes. Once this is completed, the submodel can be decomposed further into smaller models each consisting of one or more intermediate processes without degrading the overall system performance.

PEPA is later used to model Ingres Clustered DBMS and Informix XPS DBMS both running on an ICL Goldrush machine in *Chapter 7*. The study focuses on the utilisation of inter-transaction and intra-transaction parallelism. Both systems have been simplified to suit the purpose of the study. When solving the system using the decompositional evaluation approach, some submodels initially suffer from the recursive dependency on the arrival rate of requests in which the evaluation process iterates among these submodels without convergence. For this

reason, these submodels are grouped together to form a lumped submodel which transforms the dependency into internal actions.

The results obtained from the PEPA models are compared to those obtained from STEADY (analytical throughput estimator). The comparison shows that both sets of results are in good agreement for most cases. As for STEADY, its results have been calibrated and validated against those collected from Goldrush [102].

The final part of the thesis takes a closer look at the data placement. Various data placement strategies are discussed. Among them, many have been implemented in database machines such as Bubba, Gamma, Tandem and Teradata. Early research has studied the impact of data placement on the system performance of parallel DBMS but it has been confined to the placement at PE level only. Thus, we conduct a study on dual-level data placement where the data placement strategies are applied to both PE and disk levels.

9.1 Evaluation and Future Plan

A parallel database system is one of the most complicated computing systems to model. This thesis takes advantage of the features offered by PEPA to model the performance of parallel database systems. The simplicity of the language makes the models easy to understand. The compositionality of PEPA allows the components of a model to be constructed separately and yet preserves the interactions between the components. Finally, the reusability of existing specifications of the models or components makes the performance modelling task a lot more convenient and faster.

Through PEPA, this thesis has captured the behaviour of parallel database systems. It has been shown that this behaviour depends on several factors. The most significant one is the system configuration. The number of processing elements in the system has the most impact to the

system behaviour. Nevertheless, the utilisation of inter- and intra- transaction parallelism also has a significant effect on the overall system behaviour. This in turn depends on the data distribution amongst the processing elements. In order words, the performance of parallel database systems depends on the system behaviour, which in turn depends on the system configuration and data placement.

The most significant contribution of this thesis has been the introduction of decompositional evaluation approach. As we already know, PEPA is prone to the problem of state space explosion. Although model simplification and aggregation techniques (e.g. strong equivalence aggregation) may reduce the state space of a model significantly, this may not be sufficient to converge on a solution.

The decompositional evaluation approach adapts the idea of the flow equivalent aggregation technique of queueing network theory into PEPA evaluation approach and allows a model to be decomposed into submodels which are subsequently evaluated in isolation. Examples in this thesis have shown that this approach has produced identical results to those obtained by solving the PEPA model as a whole. Subsequently, experiments in *Chapter 7* have further demonstrated that the results obtained using this approach are in good agreement to those predicted by an analytical tool (STEADY).

However, if there is a recursive dependency amongst several components, these components cannot be decomposed into submodels. In contrast, these components will be grouped together to form a lumped submodel. In this case, the state space of this submodel may be too large to converge on a solution. This is the main reason why the 7 PE case in the performance modelling experiment of Ingres Cluster DBMS and Informix XPS DBMS (*Chapter 7*) failed to converge on a solution. A possible solution to this problem is to apply the compositional strong equivalence aggregation technique to these components. If this approach does not help to reduce the size of the state space sufficiently, another option to the solution will be to redistribute the data amongst the processing elements. In which case, the recursive dependency amongst certain components

may be isolated and thus allow the components to be modelled in separate submodels. This requires further research.

Currently, the parallel database system performance modelling using PEPA and following the decompositional evaluation approach is conducted manually. Nevertheless, the process of decomposing a PEPA model into submodels and subsequently evaluating the submodels in turn can be performed automatically. Therefore, it is essential for the tool to distinguish the dependency between submodels and separate them accordingly. In addition, the tool would have to be able to identify the recursive/cyclic dependency that may exist in order to guarantee convergence. It may be possible to include the tool into the PEPA workbench in the future. However, further research is required.

APPENDIX A

Lumped *ModelA*

```
#N0 = (request,r_req).N110;
#N110 = (deliver,r_dlv).N113 + (request,r_req).N90;
#N113 =(release,r_rel).N116 + (reply,r_rply).N28 + (request,r_req).N97;
#N90 = (deliver,r_dlv).N97 + (request,r_req).N46;
#N116 = (reply,r_rply).N0 + (request,r_req).N134;
#N28 = (release,r_rel).N0 + (request,r_req).N29;
#N97 = (deliver,r_dlv).N49 + (release,r_rel).N134 + (reply,r_rply).N29 +
       (request,r_req).N67;
#N46 = (deliver,r_dlv).N67 + (request,r_req).N41;
#N134 = (deliver,r_dlv).N142 + (reply,r_rply).N110 +
(request,r_req).N123;
#N29 = (deliver,r_dlv).N120 + (release,r_rel).N110 +
(request,r_req).N118;
#N49 = (release,r_rel).N142 + (reply,r_rply).N120 +
(request,r_req).N114;
#N67 = (deliver,r_dlv).N114 + (release,r_rel).N123 + (reply,r_rply).N118
      +(request,r_req).N53;
#N41 = (deliver,r_dlv).N53 + (request,r_req).N102;
#N142 = (release,r_rel).N88 + (reply,r_rply).N113 +
       (request,r_req).N124;
#N123 = (deliver,r_dlv).N124 + (reply,r_rply).N90 + (request,r_req).N38;
#N120 = (release,r_rel).N113 + (request,r_req).N129;
#N118 = (deliver,r_dlv).N129 + (release,r_rel).N90 +
       (request,r_req).N127;
#N114 = (deliver,r_dlv).N20 + (release,r_rel).N124 + (reply,r_rply).N129
       + (request,r_req).N132;
#N53 = (deliver,r_dlv).N132 + (release,r_rel).N38 + (reply,r_rply).N127
       + (request,r_req).N48;
#N102 = (deliver,r_dlv).N48 + (request,r_req).N92;
#N88 = (reply,r_rply).N116 + (request,r_req).N108;
#N124 = (deliver,r_dlv).N50 + (release,r_rel).N108 + (reply,r_rply).N97
       + (request,r_req).N126;
#N38 = (deliver,r_dlv).N126 + (reply,r_rply).N46 + (request,r_req).N44;
#N129 = (deliver,r_dlv).N136 + (release,r_rel).N97 +
       (request,r_req).N138;
#N127 = (deliver,r_dlv).N138 + (release,r_rel).N46 +
       (request,r_req).N65;
#N20 = (release,r_rel).N50 + (reply,r_rply).N136 + (request,r_req).N141;
#N132 = (deliver,r_dlv).N141 + (release,r_rel).N126 +
       (reply,r_rply).N138 + request,r_req).N91;
#N48 = (deliver,r_dlv).N91 + (release,r_rel).N44 + (reply,r_rply).N65 +
       (request,r_req).N122;
#N92 = (deliver,r_dlv).N122;
#N108 = (deliver,r_dlv).N111 + (reply,r_rply).N134 +
       (request,r_req).N115;
#N50 = (release,r_rel).N111 + (reply,r_rply).N49 + (request,r_req).N137;
#N126 = (deliver,r_dlv).N137 + (release,r_rel).N115 + (reply,r_rply).N67
       + (request,r_req).N104;
#N44 = (deliver,r_dlv).N104 + (reply,r_rply).N41 + (request,r_req).N79;
#N136 = (release,r_rel).N49 + (request,r_req).N86;
#N138 = (deliver,r_dlv).N86 + (release,r_rel).N67 + (request,r_req).N95;
#N65 = (deliver,r_dlv).N95 + (release,r_rel).N41 + (request,r_req).N103;
#N141 = (release,r_rel).N137 + (reply,r_rply).N86 + (request,r_req).N96;
#N91 = (deliver,r_dlv).N96 + (release,r_rel).N104 + (reply,r_rply).N95 +
       (request,r_req).N121;
#N122 = (deliver,r_dlv).N121 + (release,r_rel).N79 + (reply,r_rply).N103
       + (request,r_req).N131;
```

```
#N111 = (reply,r_rply).N142 + (request,r_req).N25;
#N115 = (deliver,r_dlv).N25 + (reply,r_rply).N123 +
        (request,r_req).N133;
#N137 = (release,r_rel).N25 + (reply,r_rply).N114 + (request,r_req).N74;
#N104 = (deliver,r_dlv).N74 + (release,r_rel).N133 + (reply,r_rply).N53
       + (request,r_req).N94;
#N79 = (deliver,r_dlv).N94 + (reply,r_rply).N102 + (request,r_req).N66;
#N86 = (release,r_rel).N114 + (request,r_req).N100;
#N95 = (deliver,r_dlv).N100 + (release,r_rel).N53 + (request,r_req).N47;
#N103 = (deliver,r_dlv).N47 + (release,r_rel).N102 +
        (request,r_req).N93;
#N96 = (release,r_rel).N74 + (reply,r_rply).N100 + (request,r_req).N85;
#N121 = (deliver,r_dlv).N85 + (release,r_rel).N94 + (reply,r_rply).N47 +
        (request,r_req).N139;
#N131 = (deliver,r_dlv).N139 + (release,r_rel).N66 + (reply,r_rply).N93;
#N25 = (reply,r_rply).N124 + (request,r_req).N143;
#N133 = (deliver,r_dlv).N143 + (reply,r_rply).N38 + (request,r_req).N76;
#N74 = (release,r_rel).N143 + (reply,r_rply).N132 +
       (request,r_req).N101;
#N94 = (deliver,r_dlv).N101 + (release,r_rel).N76 + (reply,r_rply).N48 +
       (request,r_req).N71;
#N66 = (deliver,r_dlv).N71 + (reply,r_rply).N92;
#N100 = (release,r_rel).N132 + (request,r_req).N59;
#N47 = (deliver,r_dlv).N59 + (release,r_rel).N48 + (request,r_req).N119;
#N93 = (deliver,r_dlv).N119 + (release,r_rel).N92;
#N85 = (release,r_rel).N101 + (reply,r_rply).N59 + (request,r_req).N140;
#N139 = (deliver,r_dlv).N140 + (release,r_rel).N71 + (reply,r_rply).N119
       + (request,r_req).N109;
#N143 = (reply,r_rply).N126 + (request,r_req).N70;
#N76 = (deliver,r_dlv).N70 + (reply,r_rply).N44 + (request,r_req).N63;
#N101 = (release,r_rel).N70 + (reply,r_rply).N91 + (request,r_req).N23;
#N71 = (deliver,r_dlv).N23 + (release,r_rel).N63 + (reply,r_rply).N122 +
       (request,r_req).N45;
#N59 = (release,r_rel).N91 + (request,r_req).N51;
#N119 = (deliver,r_dlv).N51 + (release,r_rel).N122 +
        (request,r_req).N128;
#N140 = (release,r_rel).N23 + (reply,r_rply).N51 + (request,r_req).N112;
#N109 = (deliver,r_dlv).N112 + (release,r_rel).N45 +
        (reply,r_rply).N128;
#N70 = (reply,r_rply).N104 + (request,r_req).N69;
#N63 = (deliver,r_dlv).N69 + (reply,r_rply).N79 + (request,r_req).N81;
#N23 = (release,r_rel).N69 + (reply,r_rply).N121 + (request,r_req).N56;
#N45 = (deliver,r_dlv).N56 + (release,r_rel).N81 + (reply,r_rply).N131;
#N51 = (release,r_rel).N121 + (request,r_req).N135;
#N128 = (deliver,r_dlv).N135 + (release,r_rel).N131;
#N112 = (release,r_rel).N56 + (reply,r_rply).N135;
#N69 = (reply,r_rply).N94 + (request,r_req).N84;
#N81 = (deliver,r_dlv).N84 + (reply,r_rply).N66;
#N56 = (release,r_rel).N84 + (reply,r_rply).N139;
#N135 = (release,r_rel).N139;
#N84 = (reply,r_rply).N71;
```

NO

APPENDIX B

Decomposed *ModelA*

```
::::::::::::::
Submodel BM
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (ccu2bm, infty).Q0;
#Q2 = (arrival, lambda).Q3 + (ccu2bm, infty).Q1;
#Q3 = (arrival, lambda).Q4 + (ccu2bm, infty).Q2;
#Q4 = (arrival, lambda).Q5 + (ccu2bm, infty).Q3;
#Q5 = (ccu2bm, infty).Q4;

#BM = (ccu2bm,r_sgn).(deliver,r_deliver).(bm2ccu,r_sgn0).BM;
#Model = BM <arrival> Q0;
Model



::::::::::::::
Submodel CCUa
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (tm2ccu, infty).Q0;
#Q2 = (arrival, lambda).Q3 + (tm2ccu, infty).Q1;
#Q3 = (arrival, lambda).Q4 + (tm2ccu, infty).Q2;
#Q4 = (arrival, lambda).Q5 + (tm2ccu, infty).Q3;
#Q5 = (tm2ccu, infty).Q4;

#CCUa = (tm2ccu,r_sgn).(ccu2lm,r_sgn0).CCUa ;
#Model = CCUa <tm2ccu> Q0;
Model



::::::::::::::
Submodel CCUb
::::::::::::::

#Q0 = (arrival, lambda1).Q1;
#Q1 = (arrival, lambda1).Q2 + (lm2ccu, infty).Q0;
#Q2 = (arrival, lambda1).Q3 + (lm2ccu, infty).Q1;
#Q3 = (arrival, lambda1).Q4 + (lm2ccu, infty).Q2;
#Q4 = (arrival, lambda1).Q5 + (lm2ccu, infty).Q3;
#Q5 = (lm2ccu, infty).Q4;

#C = (lm2ccu,r_sgn).( begin_trans,r_begin).C;
#D = (begin_trans,infty).(ccu2bm, r_sgn0).D;
#Model = (Q0 <> D ) <lm2ccu, begin_trans> C;
Model
```

```
::::::::::::::
Submodel CCUc
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (bm2ccu, infty).Q0;
#Q2 = (arrival, lambda).Q3 + (bm2ccu, infty).Q1;
#Q3 = (arrival, lambda).Q4 + (bm2ccu, infty).Q2;
#Q4 = (arrival, lambda).Q5 + (bm2ccu, infty).Q3;
#Q5 = (bm2ccu, infty).Q4;

#P = (bm2ccu, r_sgn).(ccu2lm0,r_sgn0).(ccu2tm,r_commit).P;
#Model = Q0 <bm2ccu> P;
Model
```

```
::::::::::::::
Submodel LMa
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (ccu2lm, infty).Q0;
#Q2 = (arrival, lambda).Q3 + (ccu2lm, infty).Q1;
#Q3 = (arrival, lambda).Q4 + (ccu2lm, infty).Q2;
#Q4 = (arrival, lambda).Q5 + (ccu2lm, infty).Q3;
#Q5 = (ccu2lm, infty).Q4;

#LMa = (ccu2lm,r_sgn).(lm2ccu,r_gnt).LMa;
#Model = LMa <ccu2lm> Q0;
Model
```

```
::::::::::::::
Submodel LMb
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (ccu2lm0, infty).Q0;
#Q2 = (arrival, lambda).Q3 + (ccu2lm0, infty).Q1;
#Q3 = (arrival, lambda).Q4 + (ccu2lm0, infty).Q2;
#Q4 = (arrival, lambda).Q5 + (ccu2lm0, infty).Q3;
#Q5 = (ccu2lm0, infty).Q4;

#LMb = (ccu2lm0,r_sgn).(release,r_rel).LMb;
#Model = LMb <ccu2lm0> Q0;
Model
```

```
::::::::::::::
Submodel TMa
::::::::::::::

#TMa = (request,r_req).(tm2ccu,r_sgn0).TMa;
TMa
```

```
::::::::::::::
Submodel TMb
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (ccu2tm, infty).Q0;
#Q2 = (arrival, lambda).Q3 + (ccu2tm, infty).Q1;
#Q3 = (arrival, lambda).Q4 + (ccu2tm, infty).Q2;
#Q4 = (arrival, lambda).Q5 + (ccu2tm, infty).Q3;
#Q5 = (ccu2tm, infty).Q4;

#TMb = (ccu2tm,r_sgn).(reply, r_reply).TMb;
#Model = TMb <ccu2tm> Q0;
Model
```

APPENDIX C

Lumped *ModelB*

```
#TCD0 = (bm2ccu,infty).TCD683 + (bm2ccu1,infty).TCD717 +
          (request,r_req).TCD528;
#TCD683 = (bm2ccu1,infty).TCD569 + (request,r_req).TCD388;
#TCD717 = (bm2ccu,infty).TCD569 + (request,r_req).TCD339;
#TCD528 = (bm2ccu,infty).TCD388 + (bm2ccu1,infty).TCD339 +
     (ccu2bm,r_sgn0).TCD573 + (ccu2bm1,r_sgn0).TCD14 +
     (request,r_req).TCD739;
#TCD569 = (bm2ccu,infty).TCD669 + (bm2ccu1,infty).TCD738 +
     (release,r_rel).TCD568 + (reply,r_rply).TCD594 +
     (request,r_req).TCD620;
#TCD388 = (bm2ccu1,infty).TCD620 + (ccu2bm,r_sgn0).TCD638 +
          (ccu2bm1,r_sgn0).TCD149 + (request,r_req).TCD734;
#TCD339 = (bm2ccu,infty).TCD620 + (ccu2bm,r_sgn0).TCD334 +
          (ccu2bm1,r_sgn0).TCD494 + (request,r_req).TCD691;
#TCD573 = (bm2ccu,infty).TCD638 + (bm2ccu1,infty).TCD334 +
          (ccu2bm1,r_sgn0).TCD0 + (request,r_req).TCD675;
#TCD14 = (bm2ccu,infty).TCD149 + (bm2ccu1,infty).TCD494 +
          (ccu2bm,r_sgn0).TCD0 + (request,r_req).TCD509;
#TCD739 = (bm2ccu,infty).TCD734 + (bm2ccu1,infty).TCD691 +
     (ccu2bm,r_sgn0).TCD675 + (ccu2bm1,r_sgn0).TCD509 +
     (request,r_req).TCD727;
#TCD669 = (bm2ccu1,infty).TCD113 + (release,r_rel).TCD668 +
     (reply,r_rply).TCD682 + (request,r_req).TCD300;
#TCD738 = (bm2ccu,infty).TCD113 + (release,r_rel).TCD741 +
     (reply,r_rply).TCD709 + (request,r_req).TCD525;
#TCD568 = (bm2ccu,infty).TCD668 + (bm2ccu1,infty).TCD741 +
     (reply,r_rply).TCD0 + (request,r_req).TCD615;
#TCD594 = (bm2ccu,infty).TCD682 + (bm2ccu1,infty).TCD709 +
     (release,r_rel).TCD0 + (request,r_req).TCD529;
#TCD620 = (bm2ccu,infty).TCD300 + (bm2ccu1,infty).TCD525 +
     (ccu2bm,r_sgn0).TCD584 + (ccu2bm1,r_sgn0).TCD742 +
     (release,r_rel).TCD615 + (reply,r_rply).TCD529 +
     (request,r_req).TCD704;
#TCD638 = (bm2ccu1,infty).TCD584 + (ccu2bm1,r_sgn0).TCD683 +
     (request,r_req).TCD503;
#TCD149 = (bm2ccu1,infty).TCD742 + (ccu2bm,r_sgn0).TCD683 +
     (request,r_req).TCD719;
#TCD734 = (bm2ccu1,infty).TCD704 + (ccu2bm,r_sgn0).TCD503 +
     (ccu2bm1,r_sgn0).TCD719 + (request,r_req).TCD658;
#TCD334 = (bm2ccu,infty).TCD584 + (ccu2bm1,r_sgn0).TCD717 +
     (request,r_req).TCD724;
#TCD494 = (bm2ccu,infty).TCD742 + (ccu2bm,r_sgn0).TCD717 +
     (request,r_req).TCD470;
#TCD691 = (bm2ccu,infty).TCD704 + (ccu2bm,r_sgn0).TCD724 +
     (ccu2bm1,r_sgn0).TCD470 + (request,r_req).TCD557;
#TCD675 = (bm2ccu,infty).TCD503 + (bm2ccu1,infty).TCD724 +
     (ccu2bm1,r_sgn0).TCD528 + (request,r_req).TCD732;
#TCD509 = (bm2ccu,infty).TCD719 + (bm2ccu1,infty).TCD470 +
     (ccu2bm,r_sgn0).TCD528 + (request,r_req).TCD635;
#TCD727 = (bm2ccu,infty).TCD658 + (bm2ccu1,infty).TCD557 +
     (ccu2bm,r_sgn0).TCD732 + (ccu2bm1,r_sgn0).TCD635 +
     (request,r_req).TCD535;
#TCD113 = (release,r_rel).TCD473 + (reply,r_rply).TCD176 +
     (request,r_req).TCD491;
#TCD668 = (bm2ccu1,infty).TCD473 + (reply,r_rply).TCD683 +
     (request,r_req).TCD305;
#TCD682 = (bm2ccu1,infty).TCD176 + (release,r_rel).TCD683 +
```

```
      (request,r_req).TCD389;
#TCD300 = (bm2ccu1,infty).TCD491 + (ccu2bm,r_sgn0).TCD276 +
      (ccu2bm1,r_sgn0).TCD273 + (release,r_rel).TCD305 +
      (reply,r_rply).TCD389 + (request,r_req).TCD681;
#TCD741 = (bm2ccu,infty).TCD473 + (reply,r_rply).TCD717 +
      (request,r_req).TCD526;
#TCD709 = (bm2ccu,infty).TCD176 + (release,r_rel).TCD717 +
      (request,r_req).TCD333;
#TCD525 = (bm2ccu,infty).TCD491 + (ccu2bm,r_sgn0).TCD520 +
      (ccu2bm1,r_sgn0).TCD475 + (release,r_rel).TCD526 +
      (reply,r_rply).TCD333 + (request,r_req).TCD660;
#TCD615 = (bm2ccu,infty).TCD305 + (bm2ccu1,infty).TCD526 +
      (ccu2bm,r_sgn0).TCD585 + (ccu2bm1,r_sgn0).TCD444 +
      (reply,r_rply).TCD528 + (request,r_req).TCD705;
#TCD529 = (bm2ccu,infty).TCD389 + (bm2ccu1,infty).TCD333 +
      (ccu2bm,r_sgn0).TCD570 + (ccu2bm1,r_sgn0).TCD15 +
      (release,r_rel).TCD528 + (request,r_req).TCD736;
#TCD584 = (bm2ccu,infty).TCD276 + (bm2ccu1,infty).TCD520 +
      (ccu2bm1,r_sgn0).TCD569 + (release,r_rel).TCD585 +
      (reply,r_rply).TCD570 + (request,r_req).TCD630;
#TCD742 = (bm2ccu,infty).TCD273 + (bm2ccu1,infty).TCD475 +
      (ccu2bm,r_sgn0).TCD569 + (release,r_rel).TCD444 +
      (reply,r_rply).TCD15 + (request,r_req).TCD725;
#TCD704 = (bm2ccu,infty).TCD681 + (bm2ccu1,infty).TCD660 +
      (ccu2bm,r_sgn0).TCD630 + (ccu2bm1,r_sgn0).TCD725 +
      (release,r_rel).TCD705 + (reply,r_rply).TCD736 +
      (request,r_req).TCD730;
#TCD503 = (bm2ccu1,infty).TCD630 + (ccu2bm1,r_sgn0).TCD388 +
      (request,r_req).TCD464;
#TCD719 = (bm2ccu1,infty).TCD725 + (ccu2bm,r_sgn0).TCD388 +
      (request,r_req).TCD544;
#TCD658 = (bm2ccu1,infty).TCD730 + (ccu2bm,r_sgn0).TCD464 +
      (ccu2bm1,r_sgn0).TCD544 + (request,r_req).TCD262;
#TCD724 = (bm2ccu,infty).TCD630 + (ccu2bm1,r_sgn0).TCD339 +
      (request,r_req).TCD740;
#TCD470 = (bm2ccu,infty).TCD725 + (ccu2bm,r_sgn0).TCD339 +
      (request,r_req).TCD637;
#TCD557 = (bm2ccu,infty).TCD730 + (ccu2bm,r_sgn0).TCD740 +
      (ccu2bm1,r_sgn0).TCD637 + (request,r_req).TCD469;
#TCD732 = (bm2ccu,infty).TCD464 + (bm2ccu1,infty).TCD740 +
      (ccu2bm1,r_sgn0).TCD739 + (request,r_req).TCD583;
#TCD635 = (bm2ccu,infty).TCD544 + (bm2ccu1,infty).TCD637 +
      (ccu2bm,r_sgn0).TCD739 + (request,r_req).TCD565;
#TCD535 = (bm2ccu,infty).TCD262 + (bm2ccu1,infty).TCD469 +
      (ccu2bm,r_sgn0).TCD583 + (ccu2bm1,r_sgn0).TCD565 +
      (request,r_req).TCD677;
#TCD473 = (release,r_rel).TCD472 + (reply,r_rply).TCD569 +
      (request,r_req).TCD507;
#TCD176 = (release,r_rel).TCD569 + (request,r_req).TCD621;
#TCD491 = (ccu2bm,r_sgn0).TCD289 + (ccu2bm1,r_sgn0).TCD598 +
      (release,r_rel).TCD507 + (reply,r_rply).TCD621 +
      (request,r_req).TCD651;
#TCD305 = (bm2ccu1,infty).TCD507 + (ccu2bm,r_sgn0).TCD277 +
      (ccu2bm1,r_sgn0).TCD272 + (reply,r_rply).TCD388 +
      (request,r_req).TCD680;
#TCD389 = (bm2ccu1,infty).TCD621 + (ccu2bm,r_sgn0).TCD639 +
      (ccu2bm1,r_sgn0).TCD144 + (release,r_rel).TCD388 +
      (request,r_req).TCD735;
#TCD276 = (bm2ccu1,infty).TCD289 + (ccu2bm1,r_sgn0).TCD669 +
      (release,r_rel).TCD277 + (reply,r_rply).TCD639 +
      (request,r_req).TCD655;
#TCD273 = (bm2ccu1,infty).TCD598 + (ccu2bm,r_sgn0).TCD669 +
      (release,r_rel).TCD272 + (reply,r_rply).TCD144 +
      (request,r_req).TCD647;
```

```
#TCD681 = (bm2ccu1,infty).TCD651 + (ccu2bm,r_sgn0).TCD655 +
      (ccu2bm1,r_sgn0).TCD647 + (release,r_rel).TCD680 +
      (reply,r_rply).TCD735 + (request,r_req).TCD752;
#TCD526 = (bm2ccu,infty).TCD507 + (ccu2bm,r_sgn0).TCD521 +
      (ccu2bm1,r_sgn0).TCD474 + (reply,r_rply).TCD339 +
      (request,r_req).TCD664;
#TCD333 = (bm2ccu,infty).TCD621 + (ccu2bm,r_sgn0).TCD332 +
      (ccu2bm1,r_sgn0).TCD495 + (release,r_rel).TCD339 +
      (request,r_req).TCD693;
#TCD520 = (bm2ccu,infty).TCD289 + (ccu2bm1,r_sgn0).TCD738 +
      (release,r_rel).TCD521 + (reply,r_rply).TCD332 +
      (request,r_req).TCD463;
#TCD475 = (bm2ccu,infty).TCD598 + (ccu2bm,r_sgn0).TCD738 +
      (release,r_rel).TCD474 + (reply,r_rply).TCD495 +
      (request,r_req).TCD101;
#TCD660 = (bm2ccu,infty).TCD651 + (ccu2bm,r_sgn0).TCD463 +
      (ccu2bm1,r_sgn0).TCD101 + (release,r_rel).TCD664 +
      (reply,r_rply).TCD693 + (request,r_req).TCD580;
#TCD585 = (bm2ccu,infty).TCD277 + (bm2ccu1,infty).TCD521 +
      (ccu2bm1,r_sgn0).TCD568 + (reply,r_rply).TCD573 +
      (request,r_req).TCD631;
#TCD444 = (bm2ccu,infty).TCD272 + (bm2ccu1,infty).TCD474 +
      (ccu2bm,r_sgn0).TCD568 + (reply,r_rply).TCD14 +
      (request,r_req).TCD590;
#TCD705 = (bm2ccu,infty).TCD680 + (bm2ccu1,infty).TCD664 +
      (ccu2bm,r_sgn0).TCD631 + (ccu2bm1,r_sgn0).TCD590 +
      (reply,r_rply).TCD739 + (request,r_req).TCD728;
#TCD570 = (bm2ccu,infty).TCD639 + (bm2ccu1,infty).TCD332 +
      (ccu2bm1,r_sgn0).TCD594 + (release,r_rel).TCD573 +
      (request,r_req).TCD674;
#TCD15 = (bm2ccu,infty).TCD144 + (bm2ccu1,infty).TCD495 +
      (ccu2bm,r_sgn0).TCD594 + (release,r_rel).TCD14 +
      (request,r_req).TCD508;
#TCD736 = (bm2ccu,infty).TCD735 + (bm2ccu1,infty).TCD693 +
      (ccu2bm,r_sgn0).TCD674 + (ccu2bm1,r_sgn0).TCD508 +
      (release,r_rel).TCD739 + (request,r_req).TCD726;
#TCD630 = (bm2ccu,infty).TCD655 + (bm2ccu1,infty).TCD463 +
      (ccu2bm1,r_sgn0).TCD620 + (release,r_rel).TCD631 +
      (reply,r_rply).TCD674 + (request,r_req).TCD648;
#TCD725 = (bm2ccu,infty).TCD647 + (bm2ccu1,infty).TCD101 +
      (ccu2bm,r_sgn0).TCD620 + (release,r_rel).TCD590 +
      (reply,r_rply).TCD508 + (request,r_req).TCD688;
#TCD730 = (bm2ccu,infty).TCD752 + (bm2ccu1,infty).TCD580 +
      (ccu2bm,r_sgn0).TCD648 + (ccu2bm1,r_sgn0).TCD688 +
      (release,r_rel).TCD728 + (reply,r_rply).TCD726 +
      (request,r_req).TCD671;
#TCD464 = (bm2ccu1,infty).TCD648 + (ccu2bm1,r_sgn0).TCD734 +
      (request,r_req).TCD702;
#TCD544 = (bm2ccu1,infty).TCD688 + (ccu2bm,r_sgn0).TCD734 +
      (request,r_req).TCD391;
#TCD262 = (bm2ccu1,infty).TCD671 + (ccu2bm,r_sgn0).TCD702 +
      (ccu2bm1,r_sgn0).TCD391 + (request,r_req).TCD451;
#TCD740 = (bm2ccu,infty).TCD648 + (ccu2bm1,r_sgn0).TCD691 +
      (request,r_req).TCD744;
#TCD637 = (bm2ccu,infty).TCD688 + (ccu2bm,r_sgn0).TCD691 +
      (request,r_req).TCD695;
#TCD469 = (bm2ccu,infty).TCD671 + (ccu2bm,r_sgn0).TCD744 +
      (ccu2bm1,r_sgn0).TCD695 + (request,r_req).TCD90;
#TCD583 = (bm2ccu,infty).TCD702 + (bm2ccu1,infty).TCD744 +
      (ccu2bm1,r_sgn0).TCD727 + (request,r_req).TCD204;
#TCD565 = (bm2ccu,infty).TCD391 + (bm2ccu1,infty).TCD695 +
      (ccu2bm,r_sgn0).TCD727 + (request,r_req).TCD142;
#TCD677 = (bm2ccu,infty).TCD451 + (bm2ccu1,infty).TCD90 +
      (ccu2bm,r_sgn0).TCD204 + (ccu2bm1,r_sgn0).TCD142;
```

```
#TCD472 = (reply,r_rply).TCD568 + (request,r_req).TCD504;
#TCD507 = (ccu2bm,r_sgn0).TCD563 + (ccu2bm1,r_sgn0).TCD597 +
      (release,r_rel).TCD504 + (reply,r_rply).TCD620 +
      (request,r_req).TCD723;
#TCD621 = (ccu2bm,r_sgn0).TCD54 + (ccu2bm1,r_sgn0).TCD743 +
      (release,r_rel).TCD620 + (request,r_req).TCD666;
#TCD289 = (ccu2bm1,r_sgn0).TCD113 + (release,r_rel).TCD563 +
      (reply,r_rply).TCD54 + (request,r_req).TCD665;
#TCD598 = (ccu2bm,r_sgn0).TCD113 + (release,r_rel).TCD597 +
      (reply,r_rply).TCD743 + (request,r_req).TCD553;
#TCD651 = (ccu2bm,r_sgn0).TCD665 + (ccu2bm1,r_sgn0).TCD553 +
      (release,r_rel).TCD723 + (reply,r_rply).TCD666 +
      (request,r_req).TCD686;
#TCD277 = (bm2ccu1,infty).TCD563 + (ccu2bm1,r_sgn0).TCD668 +
      (reply,r_rply).TCD638 + (request,r_req).TCD654;
#TCD272 = (bm2ccu1,infty).TCD597 + (ccu2bm,r_sgn0).TCD668 +
      (reply,r_rply).TCD149 + (request,r_req).TCD646;
#TCD680 = (bm2ccu1,infty).TCD723 + (ccu2bm,r_sgn0).TCD654 +
      (ccu2bm1,r_sgn0).TCD646 + (reply,r_rply).TCD734 +
      (request,r_req).TCD751;
#TCD639 = (bm2ccu1,infty).TCD54 + (ccu2bm1,r_sgn0).TCD682 +
      (release,r_rel).TCD638 + (request,r_req).TCD498;
#TCD144 = (bm2ccu1,infty).TCD743 + (ccu2bm,r_sgn0).TCD682 +
      (release,r_rel).TCD149 + (request,r_req).TCD718;
#TCD735 = (bm2ccu1,infty).TCD666 + (ccu2bm,r_sgn0).TCD498 +
      (ccu2bm1,r_sgn0).TCD718 + (release,r_rel).TCD734 +
      (request,r_req).TCD657;
#TCD655 = (bm2ccu1,infty).TCD665 + (ccu2bm1,r_sgn0).TCD300 +
      (release,r_rel).TCD654 + (reply,r_rply).TCD498 +
      (request,r_req).TCD576;
#TCD647 = (bm2ccu1,infty).TCD553 + (ccu2bm,r_sgn0).TCD300 +
      (release,r_rel).TCD646 + (reply,r_rply).TCD718 +
      (request,r_req).TCD600;
#TCD752 = (bm2ccu1,infty).TCD686 + (ccu2bm,r_sgn0).TCD576 +
      (ccu2bm1,r_sgn0).TCD600 + (release,r_rel).TCD751 +
      (reply,r_rply).TCD657 + (request,r_req).TCD371;
#TCD521 = (bm2ccu,infty).TCD563 + (ccu2bm1,r_sgn0).TCD741 +
      (reply,r_rply).TCD334 + (request,r_req).TCD458;
#TCD474 = (bm2ccu,infty).TCD597 + (ccu2bm,r_sgn0).TCD741 +
      (reply,r_rply).TCD494 + (request,r_req).TCD99;
#TCD664 = (bm2ccu,infty).TCD723 + (ccu2bm,r_sgn0).TCD458 +
      (ccu2bm1,r_sgn0).TCD99 + (reply,r_rply).TCD691 +
      (request,r_req).TCD578;
#TCD332 = (bm2ccu,infty).TCD54 + (ccu2bm1,r_sgn0).TCD709 +
      (release,r_rel).TCD334 + (request,r_req).TCD721;
#TCD495 = (bm2ccu,infty).TCD743 + (ccu2bm,r_sgn0).TCD709 +
      (release,r_rel).TCD494 + (request,r_req).TCD471;
#TCD693 = (bm2ccu,infty).TCD666 + (ccu2bm,r_sgn0).TCD721 +
      (ccu2bm1,r_sgn0).TCD471 + (release,r_rel).TCD691 +
      (request,r_req).TCD558;
#TCD463 = (bm2ccu,infty).TCD665 + (ccu2bm1,r_sgn0).TCD525 +
      (release,r_rel).TCD458 + (reply,r_rply).TCD721 +
      (request,r_req).TCD624;
#TCD101 = (bm2ccu,infty).TCD553 + (ccu2bm,r_sgn0).TCD525 +
      (release,r_rel).TCD99 + (reply,r_rply).TCD471 +
      (request,r_req).TCD652;
#TCD580 = (bm2ccu,infty).TCD686 + (ccu2bm,r_sgn0).TCD624 +
      (ccu2bm1,r_sgn0).TCD652 + (release,r_rel).TCD578 +
      (reply,r_rply).TCD558 + (request,r_req).TCD592;
#TCD631 = (bm2ccu,infty).TCD654 + (bm2ccu1,infty).TCD458 +
      (ccu2bm1,r_sgn0).TCD615 + (reply,r_rply).TCD675 +
      (request,r_req).TCD421;
#TCD590 = (bm2ccu,infty).TCD646 + (bm2ccu1,infty).TCD99 +
      (ccu2bm,r_sgn0).TCD615 + (reply,r_rply).TCD509 +
```

```
      (request,r_req).TCD689;
#TCD728 = (bm2ccu,infty).TCD751 + (bm2ccu1,infty).TCD578 +
      (ccu2bm,r_sgn0).TCD421 + (ccu2bm1,r_sgn0).TCD689 +
      (reply,r_rply).TCD727 + (request,r_req).TCD415;
#TCD674 = (bm2ccu,infty).TCD498 + (bm2ccu1,infty).TCD721 +
      (ccu2bm1,r_sgn0).TCD529 + (release,r_rel).TCD675 +
      (request,r_req).TCD733;
#TCD508 = (bm2ccu,infty).TCD718 + (bm2ccu1,infty).TCD471 +
      (ccu2bm,r_sgn0).TCD529 + (release,r_rel).TCD509 +
      (request,r_req).TCD634;
#TCD726 = (bm2ccu,infty).TCD657 + (bm2ccu1,infty).TCD558 +
      (ccu2bm,r_sgn0).TCD733 + (ccu2bm1,r_sgn0).TCD634 +
      (release,r_rel).TCD727 + (request,r_req).TCD536;
#TCD648 = (bm2ccu,infty).TCD576 + (bm2ccu1,infty).TCD624 +
      (ccu2bm1,r_sgn0).TCD704 + (release,r_rel).TCD421 +
      (reply,r_rply).TCD733 + (request,r_req).TCD678;
#TCD688 = (bm2ccu,infty).TCD600 + (bm2ccu1,infty).TCD652 +
      (ccu2bm,r_sgn0).TCD704 + (release,r_rel).TCD689 +
      (reply,r_rply).TCD634 + (request,r_req).TCD310;
#TCD671 = (bm2ccu,infty).TCD371 + (bm2ccu1,infty).TCD592 +
      (ccu2bm,r_sgn0).TCD678 + (ccu2bm1,r_sgn0).TCD310 +
      (release,r_rel).TCD415 + (reply,r_rply).TCD536 +
      (request,r_req).TCD407;
#TCD702 = (bm2ccu1,infty).TCD678 + (ccu2bm1,r_sgn0).TCD658 +
      (request,r_req).TCD692;
#TCD391 = (bm2ccu1,infty).TCD310 + (ccu2bm,r_sgn0).TCD658 +
      (request,r_req).TCD538;
#TCD451 = (bm2ccu1,infty).TCD407 + (ccu2bm,r_sgn0).TCD692 +
      (ccu2bm1,r_sgn0).TCD538;
#TCD744 = (bm2ccu,infty).TCD678 + (ccu2bm1,r_sgn0).TCD557 +
      (request,r_req).TCD152;
#TCD695 = (bm2ccu,infty).TCD310 + (ccu2bm,r_sgn0).TCD557 +
      (request,r_req).TCD318;
#TCD90 = (bm2ccu,infty).TCD407 + (ccu2bm,r_sgn0).TCD152 +
      (ccu2bm1,r_sgn0).TCD318;
#TCD204 = (bm2ccu,infty).TCD692 + (bm2ccu1,infty).TCD152 +
      (ccu2bm1,r_sgn0).TCD535;
#TCD142 = (bm2ccu,infty).TCD538 + (bm2ccu1,infty).TCD318 +
      (ccu2bm,r_sgn0).TCD535;
#TCD504 = (ccu2bm,r_sgn0).TCD562 + (ccu2bm1,r_sgn0).TCD34 +
      (reply,r_rply).TCD615 + (request,r_req).TCD722;
#TCD563 = (ccu2bm1,r_sgn0).TCD473 + (release,r_rel).TCD562 +
      (reply,r_rply).TCD584 + (request,r_req).TCD663;
#TCD597 = (ccu2bm,r_sgn0).TCD473 + (release,r_rel).TCD34 +
      (reply,r_rply).TCD742 + (request,r_req).TCD552;
#TCD723 = (ccu2bm,r_sgn0).TCD663 + (ccu2bm1,r_sgn0).TCD552 +
      (release,r_rel).TCD722 + (reply,r_rply).TCD704 +
      (request,r_req).TCD711;
#TCD54 = (ccu2bm1,r_sgn0).TCD176 + (release,r_rel).TCD584 +
      (request,r_req).TCD433;
#TCD743 = (ccu2bm,r_sgn0).TCD176 + (release,r_rel).TCD742 +
      (request,r_req).TCD720;
#TCD666 = (ccu2bm,r_sgn0).TCD433 + (ccu2bm1,r_sgn0).TCD720 +
      (release,r_rel).TCD704 + (request,r_req).TCD712;
#TCD665 = (ccu2bm1,r_sgn0).TCD491 + (release,r_rel).TCD663 +
      (reply,r_rply).TCD433 + (request,r_req).TCD748;
#TCD553 = (ccu2bm,r_sgn0).TCD491 + (release,r_rel).TCD552 +
      (reply,r_rply).TCD720 + (request,r_req).TCD706;
#TCD686 = (ccu2bm,r_sgn0).TCD748 + (ccu2bm1,r_sgn0).TCD706 +
      (release,r_rel).TCD711 + (reply,r_rply).TCD712 +
      (request,r_req).TCD632;
#TCD654 = (bm2ccu1,infty).TCD663 + (ccu2bm1,r_sgn0).TCD305 +
      (reply,r_rply).TCD503 + (request,r_req).TCD577;
#TCD646 = (bm2ccu1,infty).TCD552 + (ccu2bm,r_sgn0).TCD305 +
```

```
      (reply,r_rply).TCD719 + (request,r_req).TCD601;
#TCD751 = (bm2ccu1,infty).TCD711 + (ccu2bm,r_sgn0).TCD577 +
      (ccu2bm1,r_sgn0).TCD601 + (reply,r_rply).TCD658 +
      (request,r_req).TCD370;
#TCD498 = (bm2ccu1,infty).TCD433 + (ccu2bm1,r_sgn0).TCD389 +
      (release,r_rel).TCD503 + (request,r_req).TCD467;
#TCD718 = (bm2ccu1,infty).TCD720 + (ccu2bm,r_sgn0).TCD389 +
      (release,r_rel).TCD719 + (request,r_req).TCD546;
#TCD657 = (bm2ccu1,infty).TCD712 + (ccu2bm,r_sgn0).TCD467 +
      (ccu2bm1,r_sgn0).TCD546 + (release,r_rel).TCD658 +
      (request,r_req).TCD264;
#TCD576 = (bm2ccu1,infty).TCD748 + (ccu2bm1,r_sgn0).TCD681 +
      (release,r_rel).TCD577 + (reply,r_rply).TCD467 +
      (request,r_req).TCD487;
#TCD600 = (bm2ccu1,infty).TCD706 + (ccu2bm,r_sgn0).TCD681 +
      (release,r_rel).TCD601 + (reply,r_rply).TCD546 +
      (request,r_req).TCD685;
#TCD371 = (bm2ccu1,infty).TCD632 + (ccu2bm,r_sgn0).TCD487 +
      (ccu2bm1,r_sgn0).TCD685 + (release,r_rel).TCD370 +
      (reply,r_rply).TCD264 + (request,r_req).TCD629;
#TCD458 = (bm2ccu,infty).TCD663 + (ccu2bm1,r_sgn0).TCD526 +
      (reply,r_rply).TCD724 + (request,r_req).TCD625;
#TCD99 = (bm2ccu,infty).TCD552 + (ccu2bm,r_sgn0).TCD526 +
      (reply,r_rply).TCD470 + (request,r_req).TCD653;
#TCD578 = (bm2ccu,infty).TCD711 + (ccu2bm,r_sgn0).TCD625 +
      (ccu2bm1,r_sgn0).TCD653 + (reply,r_rply).TCD557 +
      (request,r_req).TCD593;
#TCD721 = (bm2ccu,infty).TCD433 + (ccu2bm1,r_sgn0).TCD333 +
      (release,r_rel).TCD724 + (request,r_req).TCD737;
#TCD471 = (bm2ccu,infty).TCD720 + (ccu2bm,r_sgn0).TCD333 +
      (release,r_rel).TCD470 + (request,r_req).TCD636;
#TCD558 = (bm2ccu,infty).TCD712 + (ccu2bm,r_sgn0).TCD737 +
      (ccu2bm1,r_sgn0).TCD636 + (release,r_rel).TCD557 +
      (request,r_req).TCD468;
#TCD624 = (bm2ccu,infty).TCD748 + (ccu2bm1,r_sgn0).TCD660 +
      (release,r_rel).TCD625 + (reply,r_rply).TCD737 +
      (request,r_req).TCD616;
#TCD652 = (bm2ccu,infty).TCD706 + (ccu2bm,r_sgn0).TCD660 +
      (release,r_rel).TCD653 + (reply,r_rply).TCD636 +
      (request,r_req).TCD755;
#TCD592 = (bm2ccu,infty).TCD632 + (ccu2bm,r_sgn0).TCD616 +
      (ccu2bm1,r_sgn0).TCD755 + (release,r_rel).TCD593 +
      (reply,r_rply).TCD468 + (request,r_req).TCD214;
#TCD421 = (bm2ccu,infty).TCD577 + (bm2ccu1,infty).TCD625 +
      (ccu2bm1,r_sgn0).TCD705 + (reply,r_rply).TCD732 +
      (request,r_req).TCD279;
#TCD689 = (bm2ccu,infty).TCD601 + (bm2ccu1,infty).TCD653 +
      (ccu2bm,r_sgn0).TCD705 + (reply,r_rply).TCD635 +
      (request,r_req).TCD311;
#TCD415 = (bm2ccu,infty).TCD370 + (bm2ccu1,infty).TCD593 +
      (ccu2bm,r_sgn0).TCD279 + (ccu2bm1,r_sgn0).TCD311 +
      (reply,r_rply).TCD535 + (request,r_req).TCD36;
#TCD733 = (bm2ccu,infty).TCD467 + (bm2ccu1,infty).TCD737 +
      (ccu2bm1,r_sgn0).TCD736 + (release,r_rel).TCD732 +
      (request,r_req).TCD582;
#TCD634 = (bm2ccu,infty).TCD546 + (bm2ccu1,infty).TCD636 +
      (ccu2bm,r_sgn0).TCD736 + (release,r_rel).TCD635 +
      (request,r_req).TCD564;
#TCD536 = (bm2ccu,infty).TCD264 + (bm2ccu1,infty).TCD468 +
      (ccu2bm,r_sgn0).TCD582 + (ccu2bm1,r_sgn0).TCD564 +
      (release,r_rel).TCD535 + (request,r_req).TCD676;
#TCD678 = (bm2ccu,infty).TCD487 + (bm2ccu1,infty).TCD616 +
      (ccu2bm1,r_sgn0).TCD730 + (release,r_rel).TCD279 +
      (reply,r_rply).TCD582 + (request,r_req).TCD659;
```

```
#TCD310 = (bm2ccu,infty).TCD685 + (bm2ccu1,infty).TCD755 +
      (ccu2bm,r_sgn0).TCD730 + (release,r_rel).TCD311 +
      (reply,r_rply).TCD564 + (request,r_req).TCD640;
#TCD407 = (bm2ccu,infty).TCD629 + (bm2ccu1,infty).TCD214 +
      (ccu2bm,r_sgn0).TCD659 + (ccu2bm1,r_sgn0).TCD640 +
      (release,r_rel).TCD36 + (reply,r_rply).TCD676;
#TCD692 = (bm2ccu1,infty).TCD659 + (ccu2bm1,r_sgn0).TCD262;
#TCD538 = (bm2ccu1,infty).TCD640 + (ccu2bm,r_sgn0).TCD262;
#TCD152 = (bm2ccu,infty).TCD659 + (ccu2bm1,r_sgn0).TCD469;
#TCD318 = (bm2ccu,infty).TCD640 + (ccu2bm,r_sgn0).TCD469;
#TCD562 = (ccu2bm1,r_sgn0).TCD472 + (reply,r_rply).TCD585 +
      (request,r_req).TCD547;
#TCD34 = (ccu2bm,r_sgn0).TCD472 + (reply,r_rply).TCD444 +
      (request,r_req).TCD412;
#TCD722 = (ccu2bm,r_sgn0).TCD547 + (ccu2bm1,r_sgn0).TCD412 +
      (reply,r_rply).TCD705 + (request,r_req).TCD710;
#TCD663 = (ccu2bm1,r_sgn0).TCD507 + (release,r_rel).TCD547 +
      (reply,r_rply).TCD630 + (request,r_req).TCD753;
#TCD552 = (ccu2bm,r_sgn0).TCD507 + (release,r_rel).TCD412 +
      (reply,r_rply).TCD725 + (request,r_req).TCD716;
#TCD711 = (ccu2bm,r_sgn0).TCD753 + (ccu2bm1,r_sgn0).TCD716 +
      (release,r_rel).TCD710 + (reply,r_rply).TCD730 +
      (request,r_req).TCD750;
#TCD433 = (ccu2bm1,r_sgn0).TCD621 + (release,r_rel).TCD630 +
      (request,r_req).TCD649;
#TCD720 = (ccu2bm,r_sgn0).TCD621 + (release,r_rel).TCD725 +
      (request,r_req).TCD575;
#TCD712 = (ccu2bm,r_sgn0).TCD649 + (ccu2bm1,r_sgn0).TCD575 +
      (release,r_rel).TCD730 + (request,r_req).TCD672;
#TCD748 = (ccu2bm1,r_sgn0).TCD651 + (release,r_rel).TCD753 +
      (reply,r_rply).TCD649 + (request,r_req).TCD617;
#TCD706 = (ccu2bm,r_sgn0).TCD651 + (release,r_rel).TCD716 +
      (reply,r_rply).TCD575 + (request,r_req).TCD747;
#TCD632 = (ccu2bm,r_sgn0).TCD617 + (ccu2bm1,r_sgn0).TCD747 +
      (release,r_rel).TCD750 + (reply,r_rply).TCD672 +
      (request,r_req).TCD254;
#TCD577 = (bm2ccu1,infty).TCD753 + (ccu2bm1,r_sgn0).TCD680 +
      (reply,r_rply).TCD464 + (request,r_req).TCD484;
#TCD601 = (bm2ccu1,infty).TCD716 + (ccu2bm,r_sgn0).TCD680 +
      (reply,r_rply).TCD544 + (request,r_req).TCD684;
#TCD370 = (bm2ccu1,infty).TCD750 + (ccu2bm,r_sgn0).TCD484 +
      (ccu2bm1,r_sgn0).TCD684 + (reply,r_rply).TCD262 +
      (request,r_req).TCD628;
#TCD467 = (bm2ccu1,infty).TCD649 + (ccu2bm1,r_sgn0).TCD735 +
      (release,r_rel).TCD464 + (request,r_req).TCD703;
#TCD546 = (bm2ccu1,infty).TCD575 + (ccu2bm,r_sgn0).TCD735 +
      (release,r_rel).TCD544 + (request,r_req).TCD390;
#TCD264 = (bm2ccu1,infty).TCD672 + (ccu2bm,r_sgn0).TCD703 +
      (ccu2bm1,r_sgn0).TCD390 + (release,r_rel).TCD262 +
      (request,r_req).TCD446;
#TCD487 = (bm2ccu1,infty).TCD617 + (ccu2bm1,r_sgn0).TCD752 +
      (release,r_rel).TCD484 + (reply,r_rply).TCD703 +
      (request,r_req).TCD109;
#TCD685 = (bm2ccu1,infty).TCD747 + (ccu2bm,r_sgn0).TCD752 +
      (release,r_rel).TCD684 + (reply,r_rply).TCD390 +
      (request,r_req).TCD307;
#TCD629 = (bm2ccu1,infty).TCD254 + (ccu2bm,r_sgn0).TCD109 +
      (ccu2bm1,r_sgn0).TCD307 + (release,r_rel).TCD628 +
      (reply,r_rply).TCD446;
#TCD625 = (bm2ccu,infty).TCD753 + (ccu2bm1,r_sgn0).TCD664 +
      (reply,r_rply).TCD740 + (request,r_req).TCD619;
#TCD653 = (bm2ccu,infty).TCD716 + (ccu2bm,r_sgn0).TCD664 +
      (reply,r_rply).TCD637 + (request,r_req).TCD754;
#TCD593 = (bm2ccu,infty).TCD750 + (ccu2bm,r_sgn0).TCD619 +
```

```
      (ccu2bm1,r_sgn0).TCD754 + (reply,r_rply).TCD469 +
      (request,r_req).TCD215;
#TCD737 = (bm2ccu,infty).TCD649 + (ccu2bm1,r_sgn0).TCD693 +
      (release,r_rel).TCD740 + (request,r_req).TCD745;
#TCD636 = (bm2ccu,infty).TCD575 + (ccu2bm,r_sgn0).TCD693 +
      (release,r_rel).TCD637 + (request,r_req).TCD696;
#TCD468 = (bm2ccu,infty).TCD672 + (ccu2bm,r_sgn0).TCD745 +
      (ccu2bm1,r_sgn0).TCD696 + (release,r_rel).TCD469 +
      (request,r_req).TCD91;
#TCD616 = (bm2ccu,infty).TCD617 + (ccu2bm1,r_sgn0).TCD580 +
      (release,r_rel).TCD619 + (reply,r_rply).TCD745 +
      (request,r_req).TCD238;
#TCD755 = (bm2ccu,infty).TCD747 + (ccu2bm,r_sgn0).TCD580 +
      (release,r_rel).TCD754 + (reply,r_rply).TCD696 +
      (request,r_req).TCD377;
#TCD214 = (bm2ccu,infty).TCD254 + (ccu2bm,r_sgn0).TCD238 +
      (ccu2bm1,r_sgn0).TCD377 + (release,r_rel).TCD215 +
      (reply,r_rply).TCD91;
#TCD279 = (bm2ccu,infty).TCD484 + (bm2ccu1,infty).TCD619 +
      (ccu2bm1,r_sgn0).TCD728 + (reply,r_rply).TCD583 +
      (request,r_req).TCD656;
#TCD311 = (bm2ccu,infty).TCD684 + (bm2ccu1,infty).TCD754 +
      (ccu2bm,r_sgn0).TCD728 + (reply,r_rply).TCD565 +
      (request,r_req).TCD378;
#TCD36 = (bm2ccu,infty).TCD628 + (bm2ccu1,infty).TCD215 +
      (ccu2bm,r_sgn0).TCD656 + (ccu2bm1,r_sgn0).TCD378 +
      (reply,r_rply).TCD677;
#TCD582 = (bm2ccu,infty).TCD703 + (bm2ccu1,infty).TCD745 +
      (ccu2bm1,r_sgn0).TCD726 + (release,r_rel).TCD583 +
      (request,r_req).TCD205;
#TCD564 = (bm2ccu,infty).TCD390 + (bm2ccu1,infty).TCD696 +
      (ccu2bm,r_sgn0).TCD726 + (release,r_rel).TCD565 +
      (request,r_req).TCD141;
#TCD676 = (bm2ccu,infty).TCD446 + (bm2ccu1,infty).TCD91 +
      (ccu2bm,r_sgn0).TCD205 + (ccu2bm1,r_sgn0).TCD141 +
      (release,r_rel).TCD677;
#TCD659 = (bm2ccu,infty).TCD109 + (bm2ccu1,infty).TCD238 +
      (ccu2bm1,r_sgn0).TCD671 + (release,r_rel).TCD656 +
      (reply,r_rply).TCD205;
#TCD640 = (bm2ccu,infty).TCD307 + (bm2ccu1,infty).TCD377 +
      (ccu2bm,r_sgn0).TCD671 + (release,r_rel).TCD378 +
      (reply,r_rply).TCD141;
#TCD547 = (ccu2bm1,r_sgn0).TCD504 + (reply,r_rply).TCD631 +
      (request,r_req).TCD607;
#TCD412 = (ccu2bm,r_sgn0).TCD504 + (reply,r_rply).TCD590 +
      (request,r_req).TCD715;
#TCD710 = (ccu2bm,r_sgn0).TCD607 + (ccu2bm1,r_sgn0).TCD715 +
      (reply,r_rply).TCD728 + (request,r_req).TCD749;
#TCD753 = (ccu2bm1,r_sgn0).TCD723 + (release,r_rel).TCD607 +
      (reply,r_rply).TCD648 + (request,r_req).TCD618;
#TCD716 = (ccu2bm,r_sgn0).TCD723 + (release,r_rel).TCD715 +
      (reply,r_rply).TCD688 + (request,r_req).TCD746;
#TCD750 = (ccu2bm,r_sgn0).TCD618 + (ccu2bm1,r_sgn0).TCD746 +
      (release,r_rel).TCD749 + (reply,r_rply).TCD671 +
      (request,r_req).TCD540;
#TCD649 = (ccu2bm1,r_sgn0).TCD666 + (release,r_rel).TCD648 +
      (request,r_req).TCD679;
#TCD575 = (ccu2bm,r_sgn0).TCD666 + (release,r_rel).TCD688 +
      (request,r_req).TCD265;
#TCD672 = (ccu2bm,r_sgn0).TCD679 + (ccu2bm1,r_sgn0).TCD265 +
      (release,r_rel).TCD671 + (request,r_req).TCD406;
#TCD617 = (ccu2bm1,r_sgn0).TCD686 + (release,r_rel).TCD618 +
      (reply,r_rply).TCD679 + (request,r_req).TCD195;
#TCD747 = (ccu2bm,r_sgn0).TCD686 + (release,r_rel).TCD746 +
```

```
      (reply,r_rply).TCD265 + (request,r_req).TCD368;
#TCD254 = (ccu2bm,r_sgn0).TCD195 + (ccu2bm1,r_sgn0).TCD368 +
      (release,r_rel).TCD540 + (reply,r_rply).TCD406;
#TCD484 = (bm2ccu1,infty).TCD618 + (ccu2bm1,r_sgn0).TCD751 +
      (reply,r_rply).TCD702 + (request,r_req).TCD108;
#TCD684 = (bm2ccu1,infty).TCD746 + (ccu2bm,r_sgn0).TCD751 +
      (reply,r_rply).TCD391 + (request,r_req).TCD306;
#TCD628 = (bm2ccu1,infty).TCD540 + (ccu2bm,r_sgn0).TCD108 +
      (ccu2bm1,r_sgn0).TCD306 + (reply,r_rply).TCD451;
#TCD703 = (bm2ccu1,infty).TCD679 + (ccu2bm1,r_sgn0).TCD657 +
      (release,r_rel).TCD702 + (request,r_req).TCD690;
#TCD390 = (bm2ccu1,infty).TCD265 + (ccu2bm,r_sgn0).TCD657 +
      (release,r_rel).TCD391 + (request,r_req).TCD539;
#TCD446 = (bm2ccu1,infty).TCD406 + (ccu2bm,r_sgn0).TCD690 +
      (ccu2bm1,r_sgn0).TCD539 + (release,r_rel).TCD451;
#TCD109 = (bm2ccu1,infty).TCD195 + (ccu2bm1,r_sgn0).TCD371 +
      (release,r_rel).TCD108 + (reply,r_rply).TCD690;
#TCD307 = (bm2ccu1,infty).TCD368 + (ccu2bm,r_sgn0).TCD371 +
      (release,r_rel).TCD306 + (reply,r_rply).TCD539;
#TCD619 = (bm2ccu,infty).TCD618 + (ccu2bm1,r_sgn0).TCD578 +
      (reply,r_rply).TCD744 + (request,r_req).TCD239;
#TCD754 = (bm2ccu,infty).TCD746 + (ccu2bm,r_sgn0).TCD578 +
      (reply,r_rply).TCD695 + (request,r_req).TCD376;
#TCD215 = (bm2ccu,infty).TCD540 + (ccu2bm,r_sgn0).TCD239 +
      (ccu2bm1,r_sgn0).TCD376 + (reply,r_rply).TCD90;
#TCD745 = (bm2ccu,infty).TCD679 + (ccu2bm1,r_sgn0).TCD558 +
      (release,r_rel).TCD744 + (request,r_req).TCD151;
#TCD696 = (bm2ccu,infty).TCD265 + (ccu2bm,r_sgn0).TCD558 +
      (release,r_rel).TCD695 + (request,r_req).TCD317;
#TCD91 = (bm2ccu,infty).TCD406 + (ccu2bm,r_sgn0).TCD151 +
      (ccu2bm1,r_sgn0).TCD317 + (release,r_rel).TCD90;
#TCD238 = (bm2ccu,infty).TCD195 + (ccu2bm1,r_sgn0).TCD592 +
      (release,r_rel).TCD239 + (reply,r_rply).TCD151;
#TCD377 = (bm2ccu,infty).TCD368 + (ccu2bm,r_sgn0).TCD592 +
      (release,r_rel).TCD376 + (reply,r_rply).TCD317;
#TCD656 = (bm2ccu,infty).TCD108 + (bm2ccu1,infty).TCD239 +
      (ccu2bm1,r_sgn0).TCD415 + (reply,r_rply).TCD204;
#TCD378 = (bm2ccu,infty).TCD306 + (bm2ccu1,infty).TCD376 +
      (ccu2bm,r_sgn0).TCD415 + (reply,r_rply).TCD142;
#TCD205 = (bm2ccu,infty).TCD690 + (bm2ccu1,infty).TCD151 +
      (ccu2bm1,r_sgn0).TCD536 + (release,r_rel).TCD204;
#TCD141 = (bm2ccu,infty).TCD539 + (bm2ccu1,infty).TCD317 +
      (ccu2bm,r_sgn0).TCD536 + (release,r_rel).TCD142;
#TCD607 = (ccu2bm1,r_sgn0).TCD722 + (reply,r_rply).TCD421 +
      (request,r_req).TCD602;
#TCD715 = (ccu2bm,r_sgn0).TCD722 + (reply,r_rply).TCD689 +
      (request,r_req).TCD694;
#TCD749 = (ccu2bm,r_sgn0).TCD602 + (ccu2bm1,r_sgn0).TCD694 +
      (reply,r_rply).TCD415 + (request,r_req).TCD534;
#TCD618 = (ccu2bm1,r_sgn0).TCD711 + (release,r_rel).TCD602 +
      (reply,r_rply).TCD678 + (request,r_req).TCD425;
#TCD746 = (ccu2bm,r_sgn0).TCD711 + (release,r_rel).TCD694 +
      (reply,r_rply).TCD310 + (request,r_req).TCD367;
#TCD540 = (ccu2bm,r_sgn0).TCD425 + (ccu2bm1,r_sgn0).TCD367 +
      (release,r_rel).TCD534 + (reply,r_rply).TCD407;
#TCD679 = (ccu2bm1,r_sgn0).TCD712 + (release,r_rel).TCD678 +
      (request,r_req).TCD429;
#TCD265 = (ccu2bm,r_sgn0).TCD712 + (release,r_rel).TCD310 +
      (request,r_req).TCD641;
#TCD406 = (ccu2bm,r_sgn0).TCD429 + (ccu2bm1,r_sgn0).TCD641 +
      (release,r_rel).TCD407;
#TCD195 = (ccu2bm1,r_sgn0).TCD632 + (release,r_rel).TCD425 +
      (reply,r_rply).TCD429;
#TCD368 = (ccu2bm,r_sgn0).TCD632 + (release,r_rel).TCD367 +
```

```
      (reply,r_rply).TCD641;
#TCD108 = (bm2ccu1,infty).TCD425 + (ccu2bm1,r_sgn0).TCD370 +
      (reply,r_rply).TCD692;
#TCD306 = (bm2ccu1,infty).TCD367 + (ccu2bm,r_sgn0).TCD370 +
      (reply,r_rply).TCD538;
#TCD690 = (bm2ccu1,infty).TCD429 + (ccu2bm1,r_sgn0).TCD264 +
      (release,r_rel).TCD692;
#TCD539 = (bm2ccu1,infty).TCD641 + (ccu2bm,r_sgn0).TCD264 +
      (release,r_rel).TCD538;
#TCD239 = (bm2ccu,infty).TCD425 + (ccu2bm1,r_sgn0).TCD593 +
      (reply,r_rply).TCD152;
#TCD376 = (bm2ccu,infty).TCD367 + (ccu2bm,r_sgn0).TCD593 +
      (reply,r_rply).TCD318;
#TCD151 = (bm2ccu,infty).TCD429 + (ccu2bm1,r_sgn0).TCD468 +
      (release,r_rel).TCD152;
#TCD317 = (bm2ccu,infty).TCD641 + (ccu2bm,r_sgn0).TCD468 +
      (release,r_rel).TCD318;
#TCD602 = (ccu2bm1,r_sgn0).TCD710 + (reply,r_rply).TCD279 +
      (request,r_req).TCD422;
#TCD694 = (ccu2bm,r_sgn0).TCD710 + (reply,r_rply).TCD311 +
      (request,r_req).TCD117;
#TCD534 = (ccu2bm,r_sgn0).TCD422 + (ccu2bm1,r_sgn0).TCD117 +
      (reply,r_rply).TCD36;
#TCD425 = (ccu2bm1,r_sgn0).TCD750 + (release,r_rel).TCD422 +
      (reply,r_rply).TCD659;
#TCD367 = (ccu2bm,r_sgn0).TCD750 + (release,r_rel).TCD117 +
      (reply,r_rply).TCD640;
#TCD429 = (ccu2bm1,r_sgn0).TCD672 + (release,r_rel).TCD659;
#TCD641 = (ccu2bm,r_sgn0).TCD672 + (release,r_rel).TCD640;
#TCD422 = (ccu2bm1,r_sgn0).TCD749 + (reply,r_rply).TCD656;
#TCD117 = (ccu2bm,r_sgn0).TCD749 + (reply,r_rply).TCD378;

#BM0 = (ccu2bm,infty).(deliver,r_deliver).(bm2ccu,r_sgn0).BM0;
#BM1 = (ccu2bm1,infty).(deliver,r_deliver1).(bm2ccu1,r_sgn0).BM1;
#BM = BM0 <deliver> BM1;

#Model = TCD0 <ccu2bm,ccu2bm1,bm2ccu,bm2ccu1> BM;
Model
```

APPENDIX D

Decomposed *ModelB*

```
::::::::::::::
Submodel BMs
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (ccu2bmA, infty).(ccu2bmB,infty).Q0 +
                             (ccu2bmB, infty).(ccu2bmA,infty).Q0;
#Q2 = (arrival, lambda).Q3 + (ccu2bmA, infty).(ccu2bmB,infty).Q1 +
                             (ccu2bmB, infty).(ccu2bmA,infty).Q1;
#Q3 = (arrival, lambda).Q4 + (ccu2bmA, infty).(ccu2bmB,infty).Q2 +
                             (ccu2bmB, infty).(ccu2bmA,infty).Q2;
#Q4 = (arrival, lambda).Q5 + (ccu2bmA, infty).(ccu2bmB,infty).Q3 +
                             (ccu2bmB, infty).(ccu2bmA,infty).Q3;
#Q5 = (ccu2bmA, infty).(ccu2bmB,infty).Q4 +
           (ccu2bmB, infty).(ccu2bmA,infty).Q4;


#BMa = (ccu2bmA,r_sgn).(deliver,r_deliver).(bmA2ccu,r_sgn0).BMa;
#BMb = (ccu2bmB,r_sgn).(deliver,r_deliver1).(bmB2ccu,r_sgn0).BMb;
#BM = Q0  <ccu2bmA,ccu2bmB> (BMa <deliver> BMb) ;
BM



::::::::::::::
Submodel CCUa
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (tm2ccu, infty).Q0;
#Q2 = (arrival, lambda).Q3 + (tm2ccu, infty).Q1;
#Q3 = (arrival, lambda).Q4 + (tm2ccu, infty).Q2;
#Q4 = (arrival, lambda).Q5 + (tm2ccu, infty).Q3;
#Q5 = (tm2ccu, infty).Q4;

#CCUa = (tm2ccu,r_sgn).(ccu2lm,r_sgn0).CCUa ;
#Model = CCUa <tm2ccu> Q0;
Model
```

```
::::::::::::::
Submodel CCUb
::::::::::::::

#Q0 = (arrival, lambda1).Q1;
#Q1 = (arrival, lambda1).Q2 + (lm2ccu, infty).Q0;
#Q2 = (arrival, lambda1).Q3 + (lm2ccu, infty).Q1;
#Q3 = (arrival, lambda1).Q4 + (lm2ccu, infty).Q2;
#Q4 = (arrival, lambda1).Q5 + (lm2ccu, infty).Q3;
#Q5 = (lm2ccu, infty).Q4;

#C = (lm2ccu,r_sgn).(trans_begin,r_begin).C;

#D = (trans_begin,infty).D1;
#D1 = (ccu2bmA, r_sgn0).(ccu2bmB, r_sgn0).D +
      (ccu2bmB, r_sgn0).(ccu2bmA, r_sgn0).D;

#Model = (Q0 <> D) <lm2ccu, trans_begin> C;
Model



::::::::::::::
Submodel CCUc
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (bmA2ccu, infty).Q0;
#Q2 = (arrival, lambda).Q3 + (bmA2ccu, infty).Q1;
#Q3 = (arrival, lambda).Q4 + (bmA2ccu, infty).Q2;
#Q4 = (arrival, lambda).Q5 + (bmA2ccu, infty).Q3;
#Q5 = (bmA2ccu, infty).Q4;

#K0 = (arrival, lambda1).K1;
#K1 = (arrival, lambda1).K2 + (bmB2ccu, infty).K0;
#K2 = (arrival, lambda1).K3 + (bmB2ccu, infty).K1;
#K3 = (arrival, lambda1).K4 + (bmB2ccu, infty).K2;
#K4 = (arrival, lambda1).K5 + (bmB2ccu, infty).K3;
#K5 = (bmB2ccu, infty).K4;

#P = (bmA2ccu, r_sgn).(bmB2ccu, r_sgn).P2 +
     (bmB2ccu, r_sgn).(bmA2ccu, r_sgn).P2;
#P2 = (ccu2lm0,r_sgn0).(ccu2tm,r_commit).P;
#Model = (Q0 <arrival> K0) <bmA2ccu, bmB2ccu> P;
Model



::::::::::::::
Submodel LMa
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (ccu2lm, infty).Q0;
#Q2 = (arrival, lambda).Q3 + (ccu2lm, infty).Q1;
#Q3 = (arrival, lambda).Q4 + (ccu2lm, infty).Q2;
#Q4 = (arrival, lambda).Q5 + (ccu2lm, infty).Q3;
#Q5 = (ccu2lm, infty).Q4;

#LMa = (ccu2lm,r_sgn).(lm2ccu,r_gnt).LMa;
#Model = LMa <ccu2lm> Q0;
Model
```

```
::::::::::::::
Submodel LMb
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (ccu21m0, infty).Q0;
#Q2 = (arrival, lambda).Q3 + (ccu21m0, infty).Q1;
#Q3 = (arrival, lambda).Q4 + (ccu21m0, infty).Q2;
#Q4 = (arrival, lambda).Q5 + (ccu21m0, infty).Q3;
#Q5 = (ccu21m0, infty).Q4;

#LMb = (ccu21m0,r_sgn).(release,r_rel).LMb;
#Model = LMb <ccu21m0> Q0;
Model
```

```
::::::::::::::
Submodel TMa
::::::::::::::

#TMa = (request,r_req).(tm2ccu,r_sgn0).TMa;
TMa
```

```
::::::::::::::
Submodel TMb
::::::::::::::

#Q0 = (arrival, lambda).Q1;
#Q1 = (arrival, lambda).Q2 + (ccu2tm, infty).Q0;
#Q2 = (arrival, lambda).Q3 + (ccu2tm, infty).Q1;
#Q3 = (arrival, lambda).Q4 + (ccu2tm, infty).Q2;
#Q4 = (arrival, lambda).Q5 + (ccu2tm, infty).Q3;
#Q5 = (ccu2tm, infty).Q4;

#TMb = (ccu2tm,r_sgn).(reply, r_reply).TMb;
#Model = TMb <ccu2tm> Q0;
Model
```

APPENDIX E

ModelC: System with 2 PEs, one disk each.

```
#Q0a = (tm2ccu,infty).(ccu2lm,r_sgn0).Q0a ;
#Q0b = (lm2ccu,infty).(begin_trans,r_sgn).Q0b;
#Q0 = Q0a <> Q0b;

#P0 = (begin_trans,infty).(ccu2bm, r_sgn0).P0;
#R0 = (bm2ccu,infty).(ccu2lm0,r_sgn0).(ccu2tm,r_commit).R0;
#CCU0 = Q0 <begin_trans> P0 <> R0;

#LM0a = (ccu2lm,infty).(lm2ccu,r_gnt).LM0a;
#LM0b = (ccu2lm0,infty).(release,r_rel).LM0b;
#LM0 = LM0a <> LM0b;

#BM0 = (ccu2bm,infty).(deliver,r_deliver).(bm2ccu,r_sgn0).BM0;

#TM0a = (request,r_req).(tm2ccu,r_sgn0).(rem_req,r_rmreq).TM0a;
#TM0b = (ccu2tm,infty).(rem_reply,infty).(reply,r_reply).TM0b;
#TM0 =  TM0a <> TM0b;

#PE0 = (TM0 <> BM0 <> LM0)
       <tm2ccu,ccu2tm,bm2ccu,ccu2bm,ccu2lm,ccu2lm0,lm2ccu> CCU0;


#Q1a = (tm2ccu1,infty).(ccu2lm1,r_sgn0).Q1a ;
#Q1b = (lm2ccu1,infty).(begin_trans1,r_sgn).Q1b;
#Q1 = Q1a <> Q1b;

#P1 = (begin_trans1,infty).(ccu2bm1,r_sgn0).P1;
#R1 = (bm2ccu1,infty).(ccu2lm10,r_sgn0).(ccu2tm1,r_commit).R1;
#CCU1 = Q1 <begin_trans1> P1 <> R1;

#LM1a = (ccu2lm1,infty).(lm2ccu1,r_gnt).LM1a;
#LM1b = (ccu2lm10,infty).(release1,r_rel).LM1b;
#LM1 = LM1a <> LM1b;

#BM1 = (ccu2bm1,infty).(deliver1,r_deliver1).(bm2ccu1,r_sgn0).BM1;

#TM1a = (rem_req,infty).(tm2ccu1,r_sgn0).TM1a;
#TM1b = (ccu2tm1,infty).(rem_reply,r_rmrply).TM1b;
#TM1 =  TM1a <> TM1b;

#PE1 = (TM1 <> BM1 <> LM1)
      <tm2ccu1,ccu2tm1,bm2ccu1,ccu2bm1,ccu2lm1,ccu2lm10,lm2ccu1> CCU1;

#Model = PE0 <rem_req, rem_reply> PE1;
Model
```

APPENDIX F

Submodel *TMbs* of Ingres Cluster DBMS 3 PE Case

```
#M0 = (ccu2tm0,lam0a).M1 + (rmrp1,rM01).M0a + (rmrp20,rM02).M0b +
      (rmreply1,infty).(reply,r_reply).M0 ;
#M0a = (ccu2tm0,lam0b).M0x + (ccu2tm0,lam0b1).M0c + (rmrp20,rM0a).M0d +
       (rmreply1,infty).(reply,r_reply).M0a;
#M0b = (ccu2tm0,lam0c).M0x + (ccu2tm0,lam0c1).M0e + (rmrp1,rM0b).M0d +
       (rmreply1,infty).(reply,r_reply).M0b;
#M0c = (rmrp0,rM0c1).M0a + (rmrp20,rM0c2).M0x  +
       (rmreply1,infty).(reply,r_reply).M0c;
#M0d = (ccu2tm0,lambda0).M0x ;
#M0e = (rmrp0,rM0e1).M0b + (rmrp1,rM0e2).M0x +
       (rmreply1,infty).(reply,r_reply).M0e;
#M0x = (reply,r_reply).M0;

#M1 = (ccu2tm0,lam01).M2 + (rmrp0,rM11).M0 + (rmrp1,rM12).M1a +
      (rmrp20,rM13).M1b + (rmreply1,infty).(reply,r_reply).M1;
#M1a = (reply,rpya).M0 + (rmrp0,rM1a).M0a + (rmrp20,rM1a1).M1c +
       (rmreply1,infty).(reply,r_reply).M1a;
#M1b = (reply,rpyb).M0 + (rmrp0,rM1b).M0b + (rmrp1,rM1b1).M1c +
       (rmreply1,infty).(reply,r_reply).M1b;
#M1c = (reply,rpyc).M0 + (rmrp0,rM1c).M0d;

#M2 = (ccu2tm0,lam01).M3 + (rmrp0,rM11).M1 + (rmrp1,rM12).M2a +
      (rmrp20,rM13).M2b + (rmreply1,infty).(reply,r_reply).M2;
#M2a = (reply,rpya).M1 + (rmrp0,rM1a).M1a + (rmrp20,rM1a1).M2c +
      (rmreply1,infty).(reply,r_reply).M2a;
#M2b = (reply,rpyb).M1 + (rmrp0,rM1b).M1b + (rmrp1,rM1b1).M2c +
      (rmreply1,infty).(reply,r_reply).M2b;
#M2c = (reply,rpyc).M1 + (rmrp0,rM1c).M1c;

#M3 = (rmrp0,rM30).M2 + (rmrp1,rM31).M3a + (rmrp20,rM32).M3b +
      (rmreply1,infty).(reply,r_reply).M3;
#M3a = (reply,rpya).M2 + (rmrp0,rM1a).M2a + (rmrp20,rM1a1).M3c +
       (rmreply1,infty).(reply,r_reply).M3a;
#M3b = (reply,rpyb).M2 + (rmrp0,rM1b).M2b + (rmrp1,rM1b1).M3c +
       (rmreply1,infty).(reply,r_reply).M3b;
#M3c = (reply,rpyc).M2 + (rmrp0,rM1c).M2c;


#T0 = (ccu2tm1,lam1).T1 + (rmrp0,rT0).T0a + (rmrp21,rT01).T0b;
#T0a = (ccu2tm1,lam1a).T0x + (ccu2tm1,lam1a1).T0c + (rmrp21,rT0a).T0d;
#T0b = (ccu2tm1,lam1b).T0x + (ccu2tm1,lam1b1).T0e + (rmrp0,rT0b).T0d;
#T0c = (rmrp1,rT0c).T0a + (rmrp21,rT0c1).T0x;
#T0d = (ccu2tm1,lambda1).T0x;
#T0e = (rmrp1,rT0e).T0b + (rmrp0,T0e1).T0x;
#T0x = (rmreply1,rs).T0;

#T1 = (ccu2tm1,lam11).T2 + (rmrp1,rT1).T0 + (rmrp0,rT11).T1a +
      (rmrp21,rT12).T1b;
#T1a = (rmreply1,rT1a).T0 + (rmrp1,rT1a1).T0a + (rmrp21,rT1a2).T1c;
#T1b = (rmreply1,rT1b).T0 + (rmrp1,rT1b1).T0b + (rmrp0,rT1b2).T1c;
#T1c = (rmreply1,rT1c).T0 + (rmrp1,rT1c1).T0d;

#T2 = (ccu2tm1,lam11).T3 + (rmrp1,rT1).T1 + (rmrp0,rT11).T2a +
      (rmrp21,rT12).T2b;
#T2a = (rmreply1,rT1a).T1 + (rmrp1,rT1a1).T1a + (rmrp21,rT1a2).T2c;
#T2b = (rmreply1,rT1b).T1 + (rmrp1,rT1b1).T1b + (rmrp0,rT1b2).T2c;
#T2c = (rmreply1,rT1c).T1 + (rmrp1,rT1c1).T1c;
```

```
#T3 = (rmrp1,rT3).T2 + (rmrp0,rT31).T3a + (rmrp21,rT32).T3b;
#T3a = (rmreply1,rT1a).T2 + (rmrp1,rT1a1).T2a + (rmrp21,rT1a2).T3c;
#T3b = (rmreply1,rT1b).T2 + (rmrp1,rT1b1).T2b + (rmrp0,rT1b2).T3c;
#T3c = (rmreply1,rT1c).T2 + (rmrp1,rT1c1).T2c;


#TM2 = (ccu2tm2,lambda2).TM2a;
#TM2a = (ccu2tm2,lambda2).TM2b + (rmrp20,rs0).TM2 + (rmrp21,rs1).TM2 ;
#TM2b = (ccu2tm2,lambda2).TM2c + (rmrp20,rs0).TM2a + (rmrp21,rs1).TM2a ;
#TM2c = (ccu2tm2,lambda2).TM2d + (rmrp20,rs0).TM2b + (rmrp21,rs1).TM2b ;
#TM2d = (rmrp20,rs0).TM2c + (rmrp21,rs1).TM2c ;

#Model = (M0 <rmrp0,rmrp1,rmreply1> T0) <rmrp20,rmrp21> TM2 ;
Model
```

APPENDIX G

System throughputs (tps) obtained using different combinations of data placement strategies with varying number of PEs, 2 disks per PE and 84 warehouses.

No.of PEs	Data Placement combinations				
	H/H, H/SnH, SnH/H, SnH/SnH	H/S, SnH/S	S/H, S/SnH	S/S	Max δ (%)
3	0.000451	0.000451	0.000451	0.000451	0.00
4	0.000601	0.000574	0.000601	0.000574	4.49
5	0.000742	0.000742	0.000742	0.000702	5.39
6	0.000902	0.000902	0.000902	0.000902	0.00
7	0.001053	0.001053	0.001053	0.001053	0.00
8	0.001152	0.001146	0.001146	0.001146	0.52
9	0.001270	0.001264	0.001264	0.001264	0.47
10	0.001408	0.001400	0.001400	0.001400	0.57
11	0.001580	0.001576	0.001579	0.001579	0.25
12	0.001799	0.001579	0.001799	0.001579	12.23
13	0.001812	0.001799	0.001799	0.001798	0.78
14	0.002106	0.002106	0.002106	0.002106	0.00
15	0.002106	0.002104	0.002106	0.002106	0.10
16	0.002125	0.002104	0.002106	0.002106	0.99
17	0.002515	0.002509	0.002515	0.002106	16.26
18	0.002541	0.002514	0.002515	0.002514	1.06
19	0.002541	0.002515	0.002515	0.002514	1.06
20	0.002541	0.002541	0.002541	0.002514	0.00
21	0.003159	0.003159	0.003159	0.003159	0.00
22	0.003160	0.003157	0.003159	0.003159	0.10
23	0.003160	0.003143	0.003159	0.003159	0.54
24	0.003160	0.003158	0.003159	0.003159	0.06
25	0.003160	0.003160	0.003159	0.003159	0.03
26	0.003201	0.003157	0.003160	0.003159	1.38
27	0.003201	0.003160	0.003160	0.003159	1.31
28	0.004177	0.003159	0.004177	0.003159	24.37
29	0.004177	0.004172	0.004177	0.003159	24.37
30	0.004177	0.004174	0.004177	0.003159	24.37
31	0.004177	0.004175	0.004177	0.003159	24.37
32	0.004249	0.004177	0.004177	0.004175	1.74
33	0.004249	0.004177	0.004177	0.004175	1.74
34	0.004249	0.004177	0.004177	0.004176	1.72
35	0.004249	0.004177	0.004177	0.004176	1.72
36	0.004249	0.004174	0.004249	0.004176	1.77
37	0.004249	0.004249	0.004249	0.004176	1.72
38	0.004249	0.004249	0.004249	0.004176	1.72
39	0.004249	0.004249	0.004249	0.004176	1.72
40	0.004249	0.004249	0.004249	0.004176	1.72
41	0.004249	0.004249	0.004249	0.004176	1.72
42	0.006317	0.006317	0.006317	0.006317	0.00
43	0.006319	0.006302	0.006317	0.006317	0.27
44	0.006319	0.006310	0.006317	0.006317	0.14
45	0.006319	0.006314	0.006317	0.006317	0.08
46	0.006319	0.006314	0.006317	0.006317	0.08
47	0.006319	0.006315	0.006317	0.006317	0.06
48	0.006319	0.006316	0.006317	0.006317	0.05
49	0.006319	0.006317	0.006317	0.006317	0.03
50	0.006319	0.006316	0.006317	0.006317	0.05

51	0.006319	0.006295	0.006317	0.006317	0.38
52	0.006319	0.006310	0.006317	0.006317	0.14
53	0.006319	0.006313	0.006317	0.006317	0.10
54	0.006319	0.006315	0.006317	0.006317	0.06
55	0.006319	0.006315	0.006317	0.006317	0.06
56	0.006319	0.006316	0.006318	0.006317	0.05
57	0.006319	0.006319	0.006317	0.006317	0.03
58	0.006319	0.006319	0.006317	0.006317	0.03
59	0.006319	0.006319	0.006317	0.006317	0.03
Average	*0.003937*	*0.003902*	*0.003928*	*0.003836*	*2.57*

REFERENCES

1. W.J. Anderson, "*Continuous-time Markov Chains: An Applications Oriented Approach*", Springer-Verlag, New York, 1991.

2. F. Andres, B. Bergsten, P. Borlasalamet, P. Broughton, C. Chachaty, M. Couprie, B. Finance, G. Gardarin, K. Glynn, B. Hart, S. Kellett, S. Leunig, M. Lopez, M. Ward, P. Valduriez and M. Ziane, "EDS Collaborating for a High Performance Parallel Relational Database", in *ESPRIT Conf.*, Brussels, pp. 274 – 295, Nov 1990.

3. G. Balbo, S. Donatelli and G. Franceschinis, "Understanding Parallel Program Behavior through Petri Net Models", *Journal of Parallel and Distributed Computing*, vol. 15 no. 3, pp. 171- 187, 1992.

4. G. Balbo, S. Donatelli, G. Franceschinis, A. Mazzeo, N. Mazzocca and M. Ribaudo, "On the Computation of Performance Characterisitics of Concurrent Programs using GSPNs", *Performance Evaluation*, vol. 19 no. 2-3, pp. 195 – 222, 1994.

5. F. Bause and P.S. Kritzinger, "*Stochastic Petri Nets: An Introduction to The Theory*", Vieweg, Germany, 1996.

6. B. Bergsten, M. Couprie and P. Valduriez, "Prototyping DBS3, a Shared-memory Parallel Database System," in *Proc. of Int. Conf. on Parallel and Distributed Information System*, Miami, Florida, pp. 226 – 234, December 1991.

7. J.A. Bergstra and J.W. Klop, "Algebra of Communicating Processes with Abstraction", *Theoretical Computer Science*, vol. 37, no. 1, pp. 77 – 121, May 1985.

8. M. Bernardo, R. Gorrieri and L. Donatello, "MPA: A Stochastic Process Algebra", *Technical Report UBLCS-94-10*, Laboratory of Computer Science, University of Bologna, May 1994.

9. A. Bhide, "An Analysis of Three Transaction Processing Architectures", in *Proc. of 14th Int. Conf. on VLDB*, Los Angeles, California, pp. 339 – 350, August 1988.

10. D. Bitton, "Arm Scheduling in Shadowed Disks", in *Proc. of 34th IEEE Int. Conf. on Computer*, Digest of Papers, pp. 132 – 136, March 1989.

11. D. Bitton and J. Gray, "Disk Shadowing", *in Proc. of 14th Int. Conf. on VLDB*, LA, California, pp. 331 – 338, August, 1988.

12. D. Bitton and Turbyfill, "A Retrospective on the Wisconsin Benchmark", in M. Stonebraker ed., *Reading in Database Systems*, 2nd Ed, Morgan Kaufmann Publishers, San Francisco, California, pp. 422 – 441, 1994.

13. H. Boral, "Parallelism and Data Management", in *Proc. of 3rd Int. Conf. on Data and Knowledge Bases*, Jerusalem, Israel, pp. 362 – 373, June 1988.

14. H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith and P. Valduriez, "Prototyping Bubba: A Highly Parallel Database System", *IEEE Trans. on Knowledge Data Engineering*, vol.2, no.1, pp. 4 – 24, March 1990.

15. D. Boudique and V. Fournel, "Specification of the SMART Enhanced Version", *Research Report of ESPRIT Pythagoras Project*, May 1993.

16. J. Boulos, D. Boudique, J. Dong and Y. Susanthi, "Calibration and Validation of Oracle/Goldrush SMART Models", *Technical Report of Mercury Research Project*, September 1997.

17. P. Buchholz, "On a Markovian Process Algebra", Technical Report 500/194, Informatik IV, Universität Dortmund, April 1994.

18. F. Carino and P. Kostamaa, "Exegesis of DBS/1012 and P-90: Industrial Supercomputer Database Machine", in D. Etiemble and J.C. Syre eds., in *Proc of 4th Int. Conf. on Parallel Architecture and Languages Europe*, Paris, pp. 877 – 892, June 1992.

19. S. Caselli, G. Conte and U. Malavolta, "Topology and Process Interaction in Concurrent Architectures: A GSPN Modeling Approach", *Journal of Parallel and Distributed Computing*, vol. 15 no. 3, pp. 270 – 281, 1992.

20. D.D. Chamberlin and F.B. Schmuck, "Dynamic Data Distribution (D^3) in a Shared-Nothing Multiprocessor Data Storage", in *Proc. of 18th Int. Conf. on VLDB*, Vancouver, Canada, pp. 163 – 174, August 1992.

21. K.M. Chandy, U. Herzog and L. Woo, "Parametic Analysis of Queuing Network", *IBM Journal of Research and Development*, vol. 19, no. 1, pp. 36 – 42, January 1975.

22. K.M. Chandy, U. Herzog and L. Woo, "Approximate Analysis of General Queuing Network", *IBM Journal of Research and Development*, vol. 19 , no.1, pp. 43 – 49, January 1975.

23. P.M. Chen and D.A. Patterson, "Maximising Performance in a Striped Disk Array", in *Proc. of 17th Ann. IEEE Symposium on Computer Architecture*, Seattle, WA, pp. 322 – 331, May 1990.

24. G. Chiola, G. Balbo and G. Conte, "Generalized Stochastic Petri net – A Definition at the Net Level and Its Implication", *IEEE Transactions on Software Engineering*, vol. 19, no.2, pp. 89 – 107, 1993.

25. G. Ciardo, R. German and C. Lindemann, "A Characterization of the Stochastic Process Underlying a Stochastic Petri Net", *IEEE Transactions on Software Engineering*, vol. 20 no. 7, pp. 506 – 515, 1994.

26. G. Clark and J. Hillston, "Toward Automatic Derivation of Performance Measures from PEPA Models", in J. Hillston and R. Pooley eds., *12th UK Computer and Telecommunications Performance Engineering Workshop*, The University of Edinburgh, pp. 65 – 81, September 1996.

27. E.F. Codd, "A Relational Model of Data for Large Shared Data Banks", in M. Stonebraker ed., *Reading in Database Systems*, 2nd Ed., Morgan Kaufmann Publishers, San Francisco, CA, USA, pp. 5 – 15, 1994.

28. R.B. Cooper, "*Introduction to Queueing Theory*", 2nd Ed., Edward Arnold Publishers Limited, London, 1981.

29. G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data Placement in Bubba", in *Proc. of ACM-SIGMOD Int. Conf. on Management of Data*, Chicago, USA, pp. 99 – 108, May 1988.

30. E. Dempster, N. Tomov, J. Lu, C. Pua, M.H. Williams, A. Burger, H. Taylor and P. Broughton, "Verifying a Performance Estimator for Parallel DBMS", in D. Pritchard and J. Reeve eds., *4th Int. Euro-Par Conf. on Parallel Processing*, LNCS, vol.1470, Springer Verlag, Southampton, UK, pp. 126 – 135, September 1998.

31. D.J. DeWitt, "The Wisconsin Benchmark: Past, Present and Future", in J. Gray ed., *The Benchmark Handbook for Database and Transaction Processing Systems*, 2nd Ed, Morgan Kaufmann Publichers Inc., USA, pp. 269 – 316, 1993.

32. D.J. DeWitt, B. Gerber, G. Graefe, M. Heytens, K. Kumar and M. Muralikrishna, "GAMMA: A High Performance Dataflow Database Machine", in *Proc. of 12th Int. Conf. on VLDB*, Japan, pp. 228 – 237, August 1986.

33. D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, R. Jauhari, M. Muralikrishna and A. Sharma, "A Single User Evaluation of the Gamma Database Machine", in M. Kitsuregawa and H. Tanaka eds., *Database Machines and Knowledge Base Machines*, Kluwer Academic Publishers, Boston, USA, pp. 370 – 386, 1988.

34. D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H. Hsiao and R. Rasmussen, "The GAMMA Database Machine Project", *IEEE Trans. on Knowledge Data Engineering*, vol.2, no.1, pp.44 – 63, March 1990.

35. D.J. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Communications of the ACM*, vol.35 no.6, pp. 85 – 98, June 1992.

36. D.J. DeWitt, M. Smith and H. Boral, "A Single-User Performance Evaluation of the Teradata Database Machine", in D. Gawlick, M. Haynie and A. Reuter eds., *2nd Int. Workshop on High Performance Transaction Systems*, LNCS vol.359, Springer Verlag, CA, USA, pp. 244 – 275, September 1987.

37. J. Dong, "Statistic Analysis of SmartEvaluator: A SQL Query Cost Evaluator", *CRITT/ARRIT Research Report.*

38. E.B. Dynkin, "*Markov Processes Vol. 1*", Springer-Verlag, Germany, 1965.

39. A. El-Rayes, M. Kwiatkowska and S. Minton, "Analysing Performance of Lift Systems in PEPA", in J. Hillston and R. Pooley, eds., *12th UK Computer and Telecommunications Performance Engineering Workshop*, The University of Edinburgh, pp. 83 – 100, September 1996.

40. C. Fencott, "*Formal Methods for Concurrency*", International Thomson Computer Press, U.K. 1996.

41. A. Ferscha, "A Petri Net Approach for Performance Oriented Parallel Program Design", *Journal of Parallel and Distributed Computing*, vol. 15 no. 3, pp. 188 – 206, 1992.

42. D. Freedman, "*Markov Chains*", Springer-Verlag, New York, USA, 1983.

43. S. Ghandeharizadeh and D.J. DeWitt, "A Multiuser Performance Analysis of Alternative Declustering Strategies", in *Proc. of IEEE 6th Int. Conf. on Data Engineering*, Los Angeles, CA, pp. 466 – 475, February 1990.

44. S. Ghandeharizadeh and D.J. DeWitt, "Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines", in *Proc. of 16th Int. Conf. on VLDB*, Brisbane, Australia, pp. 481 – 492, August 1990.

45. S. Gilmore and J. Hillston, "The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling", in *Proc. of 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, Vienna, pp. 353 – 368, May 1994.

46. S. Gilmore, J. Hillston, R. Holton and M. Rettelbach, "Specification in Stochastic Process Algebra for a Robot Control Problem", *Int. Journal of Production Research*, vol. 34, no. 4, pp. 1065 – 1080, April 1996.

47. N. Götz, U. Herzog, and M. Rettelbach, "TIPP – A Stochastic Process Algebra", in J. Hillston and F. Moller eds., *Proc. of the Workshop on Process Algebra and Performance Modelling*, Department of Computer Science, University of Edinburgh, May 1993.

48. G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", in *the ACM SIGMOD Int. Conf. on Management of Data*, pp. 102 – 111, Atlantic City, 1990.

49. J. Gray, B. Horst and M. Walker, "Parity Striping of Disc Arrays: Low Cost Reliable Storage with Acceptable Throughput", in *Proc. of 16th Int. Conf. on VLDB*, Brisbane, Australia, pp. 148 – 161, 1990.

50. J. Gray, ed., "*The Benchmark Handbook for Database and Transaction Processing Systems*", 2nd Ed., Morgan Kaufmann Publishers Inc., USA, 1993.

51. J. Gray and A. Reuter, "*Transaction Processing: Concepts and Techniques*", Morgan Kaufmann Publishers, San Francisco, 1993.

52. J. Hanson and A. Orooji, "Experiments with Data Access and Data Placement Strategies for Multi-Computer Database Systems", in M. Kitsuregawa and H. Tanaka eds., *Database Machines and Knowledge Base Machines*, Kluwer Academic Publishers, Boston, pp. 429 – 442, 1988.

53. J. Hillston, "*A Compositional Approach for Performance Modelling*", Cambridge University Press, 1996.

54. J. Hillston, "The Nature of Synchronisation", in U. Herzog and M. Rettelbach eds., *Proc. of 2nd Int. Workshop on Process Algebras and Performance Modelling*, Erlangen, Germany, pp. 51 – 70, November 1994.

55. J. Hillston, "Compositional Markovian Modelling Using a Process Algebra", in W. Stewart ed., *2nd Int. Workshop on Numerical Solution of Markov Chains: Computations with Markov Chains*, North Carolina, pp. 177 – 196, January 1995.

56. J. Hillston, "A Class of PEPA Models Exhibiting Product Form Solution over Submodels", *Technical Report ECS-LFCS-98-382*, Laboratory for Foundation of Computer Science, University of Edinburgh, Scotland, February 1998.

57. J. Hillston, "Exploiting Structure in Solution: Decomposing Composed Models", in C. Priami ed., *6th Annual Workshop on Process Algebra and Performance Modelling*, University degli studidi Verona, Nice, France, pp. 1 – 15, September 1998.

58. J. Hillston and N. Thomas, "Product Form Solution for a Class of PEPA Models", in *Proc. of IEEE Int. Symposium on Computer Performance and Dependability*, Durham, North Carolina, pp. 152 – 161, September 1998.

59. C.A.R. Hoare, "*Communicating Sequential Processes*", Prentice-Hall International, London, U.K., 1985.

60. D.R.W. Holton, "A PEPA Specification of an Industrial Production Cell", *Computer Journal*, vol.38, no.7, pp. 542 – 551, 1995.

61. W. Hong and M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS", *Distributed and Parallel Databases*, vol.1, pp. 9 – 32, 1993.

62. K.A. Hua and C. Lee, "An Adaptive Data Placement Scheme for Parallel Database Computer Systems", in *Proc. of 16th Int. Con. On VLDB*, Brisbane, Australia, pp. 493 – 506, August 1990.

63. Informix Software Inc., "*Informix Online Dynamic Server and Administrator Guide*", Informix Press, California, USA, 1994.

64. Ingres Corporation, "*Relational System – INGRES Database Administrator's Guide*", Release 6.4, Ingres Corporation, California, December 1991.

65. Ingres Corporation, "*Relational System – Introducing INGRES*", Release 6.4, Ingres Corporation, California, December 1991.

66. C. Jou and S.A. Smolka, "Equivalences, Congruences and Complex Axiomatization of Probabilistic Processes", in J.C.M. Baeten and J.W.Klop eds., in *Conf. On Theories of Concurrency: Unification and Extension*, Springer-Verlag, Amsterdam, Netherlands, pp.367 – 383, August 1990.

67. K. Kant, "*Introduction to Computer System Performance Evaluation*", McGraw-Hill Inc., Singapore, 1992.

68. M.Y. Kim, "Synchronized Disk Interleaving", *IEEE Transactions on Computer*, vol. C.35, no. 11, pp. 978 – 988, November 1986.

69. P.J.B. King, "*Computer and Communication Systems Performance Modelling*", Prentice-Hall Int.(UK) Ltd., U.K., 1990.

70. L. Kleinrock, "*Queueing Systems Vol. 1: Theory*", John Wiley & Sons, Canada, 1975.

71. K. Kojima, S. Torii, S. Yoshizumi, "IDP – A Main Storage Based Vector Database Processor", in M. Kitsuregawa and H. Tanaka eds., *Database Machines and Knowledge Base Machines*, Kluwer Academic Publisher, Boston, pp. 47 – 60, 1988.

72. M.D.P. Leland and W.D. Roome, "The Silicon Database Machine: Rationale, Design and Results", in M. Kitsuregawa and H. Tanaka eds., *Database Machines and Knowledge Base Machines*, Kluwer Academic Publishers, Boston, pp. 311 – 324, 1988.

73. B.P. Lester, "*The Art of Parallel Programming*", Prentice-Hall Int., Englewood Cliffs, New Jersey, 1993.

74. J. Li, J. Srivastava and D. Rotem, "CMD: A Multidimensional Declustering Method for Parallel Database Systems", in *Proc. of 18th Int. Conf. on VLDB*, Vancouver, Canada, pp. 3 – 14, August 1992.

75. L. S. Lie and G. Stiege, "Analytical Performance Evaluation of Relational Database Machines", in M. Kitsuregawa and H. Tanaka eds., *Database Machines and Knowledge Base Machines*, Kluwer Academic Publishers, Boston, pp. 401 – 414, 1988.

76. R. Marek and E. Rahm, "Performance Evaluation of Parallel Transaction Processing in Shared-Nothing Database Systems", in D. Etiemble and J.C. Syre eds., *4th Int. PARLE Conf. on Parallel Architectures and Languages*, LNCS, vol.605, Springer Verlag, Paris, France, pp.295 – 310, June 1992.

77. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli and G. Franceschinis, "*Modelling With Generalised Stochastic Petri Nets*", John Willey & Son, West Sussex, England, 1995.

78. R. Milner, "*A Calculus of Communicating Systems*", Springer-Verlag, Germany, 1980.

79. R. Milner, "*Communication and Concurrency*", Prentice-hall, U.K., 1989.

80. I. Mitrani, "*Modelling of Computer and Communication Systems*", Cambridge University Press, Cambridge, UK, 87.

81. F. Moller and C. Toft, "A Temporal Calculus for Communicating Systems", in J.C.M. Baeten and J.W. Klop eds., in *Conf. on Theories of Concurrency: Unification and Extension*, Springer-Verlag, Amsterdam, Netherlands, pg.142 - 188, August 1990.

82. F. Moller and P. Stevens, "*The Edinburgh Concurrency Workbench (Version 7)*", Laboratory for Foundations of Computer Science, University of Edinburgh, November 1994.

83. R.R. Muntz and J.C.S. Lui, "Performance Analysis of Disk Arrays Under Failure", in *Proc. of 16th Int. Conf. on VLDB*, Brisbane, Australia, pp. 162 – 173, August 1990.

84. S. Nakamura, H. Minemura, T. Minohara, K. Hakura and M.Soga, "A High Speed Database Machine HDM", in M. Kitsuregawa and H. Tanaka eds., *Database Machines and Knowledge Base Machines*, Kluwer Academic Publishers, Boston, pp. 237 – 250, 1988.

85. J. Nievergelt, H. Hinterberger and K.C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure", in M. Stonebraker ed, *Readings in Database Systems*, 2nd Ed., Morgan Kaufmann Publishers, San Francisco, California, pp. 108 – 124, 1994.

86. P. O'Neil, "*Database - Principles, Programming, Performance*", Morgan Kaufmann Inc., San Francisco, CA, 1994.

87. Oracle Corporation, "*Oracle 7 Parallel Server™: Concept and Administration*", Release 7.3, Oracle Corporation, CA, January 1996.

88. A. Osterhaug, "*Guide to Parallel Programming on Sequent Computer Systems*", Prentice-Hall, Englewood Cliff, 1989.

89. M.T. Özsu and P. Valduriez, "*Principle of Distributed Database Systems*", Prentice Hall International, New Jersey, USA, 1991.

90. Page, "A Study of a Parallel Database Machine and Its Performance: The NCR/Teradata DBC/1012", in P.M.D. Gray and R.J. Lucas eds., *Advanced Database Systems, 10th British National Conf. on Database*, LNCS, vol 618, Springer-Verlag, Aberdeen, Scotland, pp. 115 – 137, July 1992.

91. D.A. Patterson, G. Gibson and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", in M. Stonebraker ed., *Readings in Database Systems*, 2nd Ed., Morgan Kaufmann Publishers, San Francisco, California, pp. 386 – 393, 1994.

92. J.L. Peterson, "*Petri Net: Theory and The Modeling of Systems*", Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

93. F. Raab, "Overview of the TPC Benchmark C: A Complex OLTP Benchmark", in J. Gray, ed., *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufmann Publisher Inc., San Francisco, California, USA, Ch. 3, pp. 131 – 267, 1993.

94. W. Reisig, "*Petri Nets: An Introduction*", Springer-Verlag, Germany, 1985.

95. K. Salem, "Disk Striping", in *Proc. of 2nd IEEE Int. Conf. on Data Engineering*, L.A., California, pp. 336 – 342, February 1986.

96. C.H. Sauer and K.M. Chandy, "*Computer Systems Performance Modeling*", Prentice Hall, Englewood Cliffs, 1981.

97. O. Serlin, "The History of Debit Credit and The TPC", in J. Gray ed., *The Benchmark Handbook for Database and Transaction Processing Systems*, 2nd Ed., Morgan Kaufmann Publishers, San Francisco, CA, Ch. 2, pp. 21 – 130, 1993.

98. M. Stonebraker, "The Case of Shared-Nothing", *Database Engineering*, vol.9 no.1, pp. 4 – 9, March 1986.

99. M. Stonebraker, R. Katz, D. Patterson and J. Ousterhout, "The Design of XPRS", in *Proc. of 14th Int. Conf. on VLDB*, Los Angeles, CA, pp. 318 – 330, August 1988.

100. M. Stonebraker ed., "*Readings in Database Systems*" 2nd Ed., Morgan Kaufmann Publishers, San Francisco, CA, USA, 1994.

101. Tandem Database Group, "NonStopSQL, a Distributed High-Performance, High-Reliability Implementation of SQL", in D. Gawlick, M. Haynie and A. Reuter eds., *2nd Int. Workshop on High Performance Transaction Systems*, LNCS vol. 359, Springer Verlag, CA, USA, pp. 60 – 103, September 1987.

102. N. Tomov, E. Dempster, M.H. Williams, A. Burger, H. Taylor, P.J.B. King and P. Broughton, "Some Results from a New Technique for Response Time Estimation in Parallel DBMS", in P. Sloot, M. Bubak, A. Hoekstra and B. Hoertzberger eds., *Proc. of 7th Int. Conf. On High Performance Computing and Networking Europe 99*, LNCS 1593, Springer Verlag, Amsterdam, Netherlands, pp. 713 – 721, April, 1999.

103. Transaction Processing Performance Council (TPC), "*TPC Benchmark™ B: Standard Specification Revision 2.0*", CA, USA, June 1994.

104. Transaction Processing Performance Council (TPC), "*TPC Benchmark™ C: Standard Specification Revision 1.0*", CA, USA, August 1992.

105. C. Turbyfill, C. Orji and D. Bitton, "AS^{3}AP: An ANSI SQL Standard Scaleable and Portable Benchmark for Relational Database Systems", in J. Gray ed., *The Benchmark Handbook for Database and Transaction Processing Systems*, 2nd Ed., Morgan Kaufmann Publishers Inc., USA, 1993.

106. P. Valduriez, "Parallel Database Systems: Open Problems and New Issues", *Distributed and Parallel Databases*, vol.1 no.2, pp. 137 – 165, 1993.

107. P. Watson and G. Catlow, "The Architecture of ICL Goldrush MegaSERVER", in *Proc. of 13th British National Conf. on Databases (BNCOD13)*, Manchester, United Kingdom, pp. 250 – 262, 1995.

108. P. Watson and P. Townsen, "The EDS Parallel Relational Database System", in P. America ed., *Workshop on Parallel Database System*, Noordwijk, Netherlands, LNCS 503, Springer-Verlag, pp. 149 – 166, September 1991.

109. G. Weikum and P. Zabback, "Tuning of Striping Units in Disk-Array-Based File Systems", in *Proc. of IEEE 2nd Int. Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pp. 80 – 87, February 1992.

110. S. Zhou and M.H.Williams, "Data Placement in Parallel Database Systems", in M. Abdelguerfi and K. Wong eds., *Parallel Database Techniques*, IEEE Computer Society Press, in print.

111. S. Zhou, "Design of a DBMS Performance Estimator – STEADY", *Technical Report PYTH.HWU.028*, Department of CEE, Heriot-Watt University, Edinburgh, 1995.

112. S. Zhou, M.H. Williams and H. Taylor, "Practical Throughput Estimator for Parallel Database Systems", *Software Engineering Journal*, vol.11, no.4, pp. 255 – 263, July 1996.

113. S. Zhou and N. Tomov, "Some Aspects of Parallel Servers on Goldrush", *Technical Report MERCURY.HWU.003*, Department of CEE, Heriot-Watt University, Edinburgh, 1996.